Copyright
Copyright © 2019 Lattice Semiconductor Corporation. All rights reserved. This
document may not, in whole or part, be reproduced, modified, distributed, or publicly
displayed without prior written consent from Lattice Semiconductor Corporation
(“Lattice”).

Trademarks
All Lattice trademarks are as listed at www.latticesemi.com/legal. Synopsys and
Synplify Pro are trademarks of Synopsys, Inc. Aldec and Active-HDL are trademarks
of Aldec, Inc. All other trademarks are the property of their respective owners.

Disclaimers
NO WARRANTIES: THE INFORMATION PROVIDED IN THIS DOCUMENT IS "AS IS"
WITHOUT ANY EXPRESS OR IMPLIED WARRANTY OF ANY KIND INCLUDING
WARRANTIES OF ACCURACY, COMPLETENESS, MERCHANTABILITY,
NONINFRINGEMENT OF INTELLECTUAL PROPERTY, OR FITNESS FOR ANY
PARTICULAR PURPOSE. IN NO EVENT WILL LATTICE OR ITS SUPPLIERS BE
LIABLE FOR ANY DAMAGES WHATSOEVER (WHETHER DIRECT, INDIRECT,
SPECIAL, INCIDENTAL, OR CONSEQUENTIAL, INCLUDING, WITHOUT
LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR
LOSS OF INFORMATION) ARISING OUT OF THE USE OF OR INABILITY TO USE
THE INFORMATION PROVIDED IN THIS DOCUMENT, EVEN IF LATTICE HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME
JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF CERTAIN
LIABILITY, SOME OF THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

Lattice may make changes to these materials, specifications, or information, or to the
products described herein, at any time without notice. Lattice makes no commitment to
update this documentation. Lattice reserves the right to discontinue any product or
service without notice and assumes no obligation to correct any errors contained
herein or to advise any user of this document of any correction if such be made. Lattice
recommends its customers obtain the latest version of the relevant information to
establish that the information being relied upon is current and before ordering any
products.
**Type Conventions Used in This Document**

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning or Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Items in the user interface that you select or click. Text that you type into the user interface.</td>
</tr>
<tr>
<td><code>&lt;Italic&gt;</code></td>
<td>Variables in commands, code syntax, and path names.</td>
</tr>
<tr>
<td><strong>Ctrl+L</strong></td>
<td>Press the two keys at the same time.</td>
</tr>
<tr>
<td><strong>Courier</strong></td>
<td>Code examples. Messages, reports, and prompts from the software.</td>
</tr>
<tr>
<td>. . .</td>
<td>Omitted material in a line of code.</td>
</tr>
<tr>
<td>.</td>
<td>Omitted lines in code and report examples.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional items in syntax descriptions. In bus specifications, the brackets are required.</td>
</tr>
<tr>
<td>( )</td>
<td>Grouped items in syntax descriptions.</td>
</tr>
<tr>
<td>{ }</td>
<td>Repeatable items in syntax descriptions.</td>
</tr>
<tr>
<td></td>
<td>A choice between items in syntax descriptions.</td>
</tr>
</tbody>
</table>
Contents

Introduction  8
Radiant Software Overview  8
User Guide Organization  9

Getting Started  10
Prerequisites  10
Creating a New Project  11
Opening an Existing Project  14
Importing a Lattice Diamond Project  15
Next Steps  15

Design Environment Fundamentals  17
Overview  17
Project-Based Environment  18
Process Flow  19
Shared Memory  20
Context-Sensitive Data Views  21
Cross-Probing  22

User Interface Operation  25
Overview  25
Start Page  26
  Menus and Toolbars  27
  Project Views  28
  Tool View Area  29
  Output and Tcl Console  30
  Basic UI Controls  30
File List  30
Source Template  31
CONTENTS

IP Catalog 33
Process 33
Task Detail View 34
Hierarchy 34
Reports 35
Tool Views 36
Tcl Console 36
Output 37
Message 37
Find Results 38
Common Tasks 38
  Controlling Views 38
  Cross-Probing Between Views 40
  Cross-Probing of the Data Path 41
  Cross-probing in Encrypted Design 43

Working with Projects 44
Overview 44
Implementations 46
  Input Files 48
    Pre-Synthesis Constraint Files 49
    Post-Synthesis Constraint Files 49
  Debug Files 50
  Script Files 51
  Analysis Files 51
  Programming Files 51
Strategies 52
  Area 53
  Quick 54
  Timing 55
  User-Defined 56
Common Tasks 56
  Creating a Project 56
  Changing the Target Device 56
  Setting the Top Level of the Design 57
  Editing Files 57
  Saving Project Data 58

Radiant Software Design Flow 59
Overview 59
  Design Flow Processes 60
  Running Processes 60
IP Encryption Flow 61
Implementation Flow and Tasks 66
  Synthesis Constraint Creation 67
  Constraint Creation 68
Simulation Flow 70
Simulation Wizard Flow 71
Working with Tools and Views  74
Overview  74
  Shared Memory  74
  Cross Probing  74
View Menu Highlights  75
Start Page  75
Reports  75
Tools  77
  Timing Constraint Editor  77
  Constraint Propagation  79
  Device Constraint Editor  79
  SSO Analysis  80
  Netlist Analyzer  80
  Floorplan View  81
  IO Planner View  83
  Physical View  83
  Timing Analysis View  84
  Reveal Inserter  85
  Reveal Analyzer  86
  Reveal Controller  86
  Power Calculator  86
  ECO Editor  88
  Programmer  89
  Run Manager  90
  Synplify Pro for Lattice  90
  Active-HDL Lattice Edition  91
  Simulation Wizard  92
  Source Template  92
  IP Catalog  92
  IP Packager  93
Common Tasks  94
  Controlling Tool Views  94
  Using Zoom Controls  96
  Displaying Tool Tips  97
  Setting Display Options  98

Command Line Reference Guide  99
Command Line Program Overview  99
Command Line Basics  101
  Command Line Data Flow  101
  Command Line General Guidelines  102
  Command Line Syntax Conventions  103
  Setting Up the Environment to Run Command Line  104
  Invoking Core Tool Command Line Programs  105
  Invoking Core Tool Command Line Tool Help  105
Command Line Tool Usage  106
  Running cmpl_libs.tcl from the Command Line  107
  Running HDL Encryption from the Command Line  109
  Running SYNTHESIS from the Command Line  116
  Running Postsyn from the Command Line  122
  Running MAP from the Command Line  123
Introduction

Lattice Radiant™ software is the leading-edge software design environment for cost-sensitive, low-power Lattice Field Programmable Gate Arrays (FPGA) architectures. The Radiant software integrated tool environment provides a modern, comprehensive user interface for controlling the Lattice Semiconductor FPGA implementation process. Its combination of new and enhanced features allows users to complete designs faster, more easily, and with better results than ever before.

This user guide describes the main features, usage, and key concepts of the Radiant software design environment. It should be used in conjunction with the Release Notes and reference documentation included with the product software. The Release Notes document is also available on the Lattice Web site and provides a list of supported devices.

Radiant Software Overview

The Radiant software uses an expanded project-based design flow and integrated tool views so that design alternatives and what-if scenarios can easily be created and analyzed. The Implementations and Strategies concepts provide a convenient way for users to try alternate design structures and manage multiple tool settings.

System-level information—including process flow, hierarchy, and file lists—is available, along with integrated HDL code checking and consolidated reporting features.

A fast Timing Analysis loop and Programmer provide capabilities in the integrated framework. The cross-probing feature and the shared memory architecture ensure fast performance and better memory utilization.

The Radiant software is highly customizable and provides Tcl scripting capabilities from either its built-in console or from an external shell.

The Radiant software has many of the same features as Lattice Diamond software, and adds new features, such as:

- Constraints support utilizing industry standard SDC format.
- Efficient, easy-to-use integrated graphical user interface (GUI) with a new look-and-feel that gives users more efficient access to popular tools.
Unified timing analysis engine with enhanced timing reports for faster design timing closure.

**User Guide Organization**

This user guide contains all the basic information for using the Radiant software. It is organized in a logical sequence from introductory material, through operational descriptions, to advanced topics.

Key concepts and work flows are explained in "Design Environment Fundamentals" on page 17 and "Radiant Software Design Flow" on page 59.

Basic operation of the design environment is described in "User Interface Operation" on page 25.

The chapter "Working with Projects" on page 44 shows how to set up project implementations and strategies.

The chapter "Working with Tools and Views" on page 74 describes the many tool views available.
Chapter 2

Getting Started

This chapter explains how to run the Radiant software and open or create a project. For more information about project fundamentals, see the chapters “Design Environment Fundamentals” on page 17 and “Working with Projects” on page 44.

Prerequisites

To run the Radiant software, select Radiant Software from the installation location. This opens the default Start Page, shown in Figure 1.

Figure 1: Default Start Page
Creating a New Project

A project is a collection of all files necessary to create and download your design to the selected device. The New Project wizard guides you through the steps of specifying a project name and location, selecting a target device, and adding existing sources to the new project.

Note
Do not place more than one project in the same directory.

To create a new project:
1. From the Radiant main window, click the New Project button, or choose File > New > Project.
   The New Project confirmation window opens, shown in Figure 2.

Figure 2: New Project Confirmation Window

2. Click Next. The New Project wizard opens, shown in Figure 3.
3. In the Project Name dialog box, do the following:
   - Under Project, specify the name for the new project.
     File names for Radiant software projects and project source files must start with a letter (A-Z, a-z) and must contain only alphanumeric characters (A-Z, a-z, 0-9) and underscores (_). Spaces are allowed.
     - To specify a location for your project, click Browse. In the Project Location dialog box, you can specify a desired location.
     - Under Implementation, specify the name for the first version of the project. You can have more than one version, or “implementation," of the project to experiment with. For more information on implementations, refer to “Implementations” on page 46.
     - To create a sub-directory with the same name as your location directory, click Create Subdirectory. This will allow you to keep your project implementations separate. If this box is left unchecked, no sub-directory will be created in the project directory.
   - When you finish, click Next.

4. In the Add Source dialog box, do the following if you have an existing source file that you want to add to the project. If there are no existing source files, click Next.
   a. Click Add Source. You can import HDL files at this time. In the Import File dialog box, browse for the source file you want to add, select it, and click Open.
      The source file is then displayed in the Source files field.
   b. Repeat the above to add more files.
   c. To copy the added source files to the implementation directory, select Copy source to implementation directory. If you prefer to reference these files, clear this option.
d. To create empty Lattice Design Constraint (.ldc) file and Physical Constraint File (.pdc) files that can be edited at a later time, select **Create empty constraint files**. Refer to the chapter “Implementations” on page 46 for more information about constraint files.

e. When you finish, click **Next**.

5. In the Select Device dialog box, shown in Figure 4, select a device family and a specific device within that family. Then choose the options you want for that device. When you finish, click **Next**.

**Figure 4: Select Device Dialog Box**

6. In the Select Synthesis Tool dialog box, select the synthesis tool that you want to use. This choice can be changed at any time. When you finish, click **Next**.

7. In the Project Information dialog box, make sure the project settings are correct.

**Note**

If you want to change some of the settings, click **Back** to modify them in the previous dialog boxes of the New Project Wizard.

Click **Finish**. The newly created project, shown in Figure 5, is now created and open.
Opening an Existing Project

Use one of the following methods to open an existing Radiant software project:

- On the Start Page, click the **Open Project** \(\mathbb{S}\) button.
- From the File menu, choose **Open > Project**.
- On the Start Page, select the desired project from the Recent Projects List. Alternatively, choose a recent project from the **File > Recent Projects** menu.

You can use the Options dialog box to increase the number of projects that are shown in the Recent Projects list and to automatically load the previous project at startup. Choose **Tools > Options** to open the dialog box. To increase the number of recent projects listed, click the **General** tab and enter a number for “Maximum items shown in Recent Project List” (up to 32). To automatically open the previous project during startup, click the **Startup** tab.
and then choose **Open Previous Project** from the “At Lattice Radiant Software startup” menu.

![Figure 6: Recent Project List](image)

**Figure 6: Recent Project List**

The file browser applies an ".ldf" file filter to help you find Lattice Diamond project files. The Lattice Diamond project is converted to a Radiant software project.

For more information about importing Lattice Diamond projects into the Radiant software, refer to the *Lattice Radiant Software Guide for Diamond Users*.

**Importing a Lattice Diamond Project**

To import a Lattice Diamond project into the Radiant software, choose **File > Open > Import Diamond Project**.

![Figure 7: Import Lattice Diamond Project](image)

**Figure 7: Import Lattice Diamond Project**

Next Steps

After you have a project opened in the Radiant software, you can go sequentially through the rest of this user guide to learn how to work with the entire design environment, or you can go directly to any topic of interest.

- The chapters “Design Environment Fundamentals” on page 17 and “Radiant Software Design Flow” on page 59 provide explanations of key concepts.
- “User Interface Operation” on page 25 provides descriptions of the functions and controls that are available in the Radiant software environment.
- The chapters “Working with Projects” on page 44 and “Working with Tools and Views” on page 74 explain how to run processes and use the design tools.
“Tcl Command Reference Guide” on page 150 provides an introduction to the scripting capabilities available, plus command-line shell examples.

“Advanced Topics” on page 186 provides further details about environment options, shared memory, and Tcl scripting.
This chapter provides background and discussion on the technology and methodology underlying the Radiant software design environment. Important key concepts and terminology are defined.

Overview

Understanding some of the fundamental concepts behind the Radiant software framework technology will increase your proficiency with the tool and allow you to quickly come up to speed on its use.

The Radiant software is a next-generation software design environment that uses a new project-based methodology. A single project can contain multiple implementations and strategies to provide easily managed alternate design structures and tool settings.

The process flow is managed at a system level with run management controls and reporting. Context-sensitive views ensure that you only see the data that is available for the current state in the process flow.

The shared memory technology enables many of the advanced functions in the Radiant software. Easy cross-probing between tool views and faster process loops are among the benefits.

Note

By loading the Radiant software multiple times, you can run different Radiant projects simultaneously. However, you must not load the same project in more than one Radiant software instance, as software conflicts can occur.

The Radiant software can also be run remotely. Refer to the Lattice Radiant Software Installation Guide for Windows or Lattice Radiant Software Installation Guide for Linux for more information.
Project-Based Environment

A project in the Radiant software consists of the following file types:

- HDL source files
- Constraint files
- Reveal debug files
- Script files for simulation
- Analysis files for power calculation and timing analysis
- Programming files

The Radiant software also includes settings for the targeted device and the different tools. The project data is organized into implementations, which define the project structural elements, and strategies, which are collections of tool settings.

The following File List shows the items in a sample project.

Figure 8: File List

Each item that is displayed in **bold** means that it has been selected as the active item for an implementation. An implementation displayed in **bold** means that it has been selected as the currently active implementation for the project. Your project must have one active implementation, and the implementation must have one active strategy. Optional items, such as Reveal hardware debugger files, can be set as active or inactive.

The project is the top-level organizational element in the Radiant software, and it can contain multiple implementations and multiple strategies. This enables you to try different design approaches within the same project. If you want to have a Verilog version of your design, for example, make an implementation that consists of only the Verilog source files. If you want another version of the design with primarily Verilog files but a Structural Verilog (.vm) netlist for one module, create a new implementation using the Verilog and .vm source files. Each implementation can have Verilog, VHDL or Structural Verilog source or mixed of them. The same project and design is used, but with a different set of modular blocks.

Similarly, if you want to try different implementation tool options, you can create a new strategy with the new option values.
You manage these multiple implementations and strategies for your project by setting them as active. There can only be a single active implementation with its one active strategy at a time.

**Process Flow**

A process is a specific task in the overall processing of a source or project. Typical processing tasks include synthesizing, mapping, placing, and routing. You can view the available processes for a design in the Process Toolbar.

![Process Toolbar](image)

Click the Task Detail View button to see detailed information of the processes.

Processes are grouped into categories according to their functions.

- **Synthesize Design**
  
  Click on this process and Lattice Synthesis Engine (LSE) runs to synthesize the design. By default, this process runs the LSE tool.

  If you are using Synplify Pro, choose Synplify Pro as the synthesis tool (Project > Active Implementation > Select Synthesis Tool).

- **Post-Synthesis Timing Analysis**

  Runs timing analysis after the Synthesize Design process.

- **Post-Synthesis Simulation File**

  Generates a netlist file `<file_name>_syn.vo` used for functional verification.

- **Map Design**

  This process maps a design to an FPGA. Map Design is the process of converting a design represented as a network of device-independent components (such as gates and flip-flops) into a network of device-specific components (for example, configurable logic blocks).

- **Map Timing Analysis**

  Runs timing analysis after the Map Design process.

- **Place & Route Design**

  After a design has undergone the necessary translation to bring it into the Unified Database (.udb) format, you can run the Place & Route Design process. This process takes a mapped physical design .udb file, places and routes the design, and then outputs a file that can then be processed by the design implementation tools.

- **Place & Route Timing Analysis**
Runs timing analysis after Place & Route process.

► **I/O Timing Analysis**

Runs I/O timing analysis that allows you to view the path delay tables and Timing Analysis View report of your timing constraints after placement and routing.

► **Export Files**

You can check the desired file you want to export and run this process.

► **Bitstream File**

This process takes a fully routed physical design as input and produces a configuration bitstream (bit images). The bitstream file contains all of the configuration information from the physical design defining the internal logic and interconnections of the FPGA, as well as device-specific information from other files associated with the target device.

► **IBIS Model**

This process generates a design-specific IBIS (I/O Buffer Information Specification) model file (<project_name>.ibs).

IBIS models provide a standardized way of representing the electrical characteristics of a digital IC’s pins (input, output, and I/O buffers).

► **Gate-Level Simulation File**

This process backannotates the routed design with timing information so that you may run a simulation of your design. The backannotated design is a Verilog netlist.

The Reports view allows you to examine and print process reports.

Messages are displayed in the Messages window at the bottom of the Radiant software main window.

The process status icons are defined as follows:

► Process in initial state (not processed)

✔ Process completed successfully

✔ Process completed with unprocessed subtask(s)

✖ Process failed

---

**Shared Memory**

The Radiant software uses a shared memory architecture. All tool and data views look at the same design data at any point in time. This means that when you change a data element in one view of your design, all other views will see the change, whether they are active or not.

When project data has been changed but not yet saved, an asterisk (*) is displayed in the title tab of the view.
Notice that the asterisks indicating changed data will appear in all views referencing the changed data.

If a tool view becomes unavailable, the Radiant software environment will need to be closed and restarted.

Context-Sensitive Data Views

The data in shared memory reflects the state or context of the overall process flow. This means that views such as Device Constraint Editor Spreadsheet View will display only the data that is currently available, depending on process steps that have been completed.

For example, Figure 11 shows that the Process flow before Synthesis. Therefore, Spreadsheet View shows no IO Type or PULLMODE.

After Synthesis has been completed, Spreadsheet View displays IO Type and PULLMODE assignments, as shown in Figure 12.
When you see the “Loading Data” message displayed in Figure 13, it means that a process has been completed and that the shared memory is being updated with new data.

All tool views are dynamically updated when new data becomes available. This means that when you rerun an earlier process while a view is open and displaying data, the view will remain open but dimmed because its data is no longer available.

Cross-Probing

Cross-probing is a feature found in most tool views, and allows common data elements to be viewed in multiple tool views. For example, in Physical View, you can right-click on a component and choose Show in > Floorplan View, as shown in Figure 14, to display the Floorplan View.
To auto cross-probe between Floorplan View and Physical View, ensure both views are attached to the Radiant software main window and then right-click on the Floorplan View tab and select **Split Tab Group**. The two views display in parallel, as shown in Figure 15.

When both Floorplan View and Physical View are open, an item that you select in one of these views is automatically selected in the other. Cross probing is especially useful for immediately examining connections from two different views.
Both tabs are merged back into a single group by right-clicking on the **Floorplan View** tab and choosing **Move to Another Tab Group**.
Chapter 4

User Interface Operation

This chapter describes the user interface features, controls, and basic operation of the Radiant software. Each major element of the interface is explained. The last section in the chapter describes common user interface tasks.

Overview

The Radiant software user interface (UI) provides a comprehensive, integrated tool environment. The UI is very flexible and configurable, enabling you to store constraints for the layout you choose.

This chapter will take you through the operation of the main elements of the UI, but you should also explore the controls at your own pace. Figure 16 shows the Radiant software main window in the default state.
Start Page

The Start Page contains three major sections, as shown in Figure 16.

- **Project**: This section allows you to create a new project; open an existing Project, and open an example.

- **Information Center**: This section has links to Getting Started, Tutorials, User Guides, and Support Center.

- **Recent Project List**: Provides a quick way to load a recent project you’ve been working on.

The Start Page appears in the View area by default when the Radiant software is first launched, and can be opened from the View tab on the menu.

The Start Page can be closed, opened, detached, and attached using the Attach button. See “Basic UI Controls” on page 30.
Menus and Toolbars

At the top of the main window is the menu and toolbar area. High-level controls for accessing tools, managing files and projects, and controlling the layout are contained here. All toolbar functionality is also contained in the menus. The menus also have functions for system, project and toolbar control.

Figure 17: Menu and Toolbar Area

The toolbars are organized into functional sets. The display of each toolbar is controlled in the View > Toolbar menu and also by right-clicking in the Menu and Toolbar area. The Process Toolbar lists all the processes available, such as Synthesize Design, Map Design, Place & Route Design, and Export Files. A process is a specific task in the overall processing of a source or project. You can view the available processes for a design in the Process Toolbar.

Click Task Detail View to see detailed information of the processes available.

The Reports view allows you to examine and print process reports. There are two panes in the Reports view. The left pane lists the reports. The right pane displays the reports.

Log messages are displayed in the Output frame of the Radiant software main window.

Figure 18: Report and Log Message Views
Project Views

In the middle of the main window on the left side is the Project View area. This is where the overall project and process flow is displayed and controlled.

Figure 19: Project View Area

Tabs at the bottom of the Project View allow you to select between the following views:

- **File List** – shows the files in the project organized by implementations and strategies. This is not a hierarchical listing of the design.
- **Source Template** – provides templates for creating VHDL, Verilog, and Constraint files.
- **IP Catalog** – lists available Modules/Intellectual Properties (IP)

Underneath the Project View is the Hierarchy View area. It allows you to view the hierarchical design representation. Hierarchy view shares the left pane with File List view.
Tool View Area

In the middle of the main window on the right side is the Tool View area. This is where the Start Page, Reports View and all the Tool views are displayed.

Figure 20: Tool View Area

Multiple tools can be displayed at the same time. The Window toolbar includes controls for grouping the tool views as well as integrating all tool views back into the main window.

Each tool view is specific to its tool and can contain additional toolbars and multiple panes or windows controlled by additional tabs. The chapter "Working with Tools and Views" on page 74 provides more details about each tool and view.

Figure 21: Multiple Tools
Output and Tcl Console

Near the right bottom of the main window is the Tcl Console, Output and Message area.

Tabs at the bottom of this area allow you to select between Tcl Console, Output, and Message. Tool output is automatically displayed in the Output tab, and Errors and Warnings in the Message tab.

![Figure 22: Output and Tcl Console Area](image)

Basic UI Controls

The Radiant software environment is based on modern industry standard user interface concepts. The menus, toolbars, and mouse pointer all behave in familiar ways. You can resize any of the window panes, drag and drop elements, right-click a design element to see available actions, and hold the mouse pointer over an object to view the tool tip.

File List

The File List is a project view that shows the files in the project, including implementations and strategies. It is not a hierarchical listing of the design, but rather a list of all the design source, configuration and control files that make up the project.
At the top level in the File List is the project name. Directly below the project name is the target device, followed by the strategies, and then the implementations. There must be one active implementation, and it must have one active strategy. Active elements are indicated in **bold**.

You can right-click any file or item in the File List to access a pop-up menu of currently available actions for that item. The pop-up menu contents vary, depending on the type of item selected.

The File List view can be hidden by clicking the small arrow in right hand side “Click to show/hide side panel”. See “Basic UI Controls” on page 30.

---

**Source Template**

The Source Template is a project view that provides templates for creating VHDL, Verilog, and constraint files. Templates increase the speed and accuracy of design entry. You can drag and drop a template directly to the source file. You can also create your own templates.

To access templates, choose **View > Show Views > Source Template**, or click **icon** in the toolbar, or click **Source Template** tab in bottom left pane, to locate and access the following templates:

- Verilog, including common and Parameterized Module Instantiation (PMI), Primitives, Attributes, Encryption, and User Templates
- VHDL, including common, PMI, Primitives, Attributes, Encryption, and User Templates
Constraints for LSE, including Timing and Physical constraints and User Templates

NOTE
For more information on PMI, refer to the Radiant software Online help. See the topic User Guides > Entering the Design > Designing with Soft IP, Modules, and PMI > PMI or IP Catalog?

You can simply drag any template and drop it into your source file.

Figure 24: Source Template

![Figure 24: Source Template](image)
**IP Catalog**

IP Catalog enables you to customize a collection of functional blocks from Lattice Semiconductor. Through the IP Catalog, you can access two types of functional blocks, Modules and IP.

To access IP catalog, choose View > Show Views > IP Catalog, or click icon in the toolbar, or click on IP Catalog tab in bottom left pane.

![Figure 25: IP Catalog](image)

Each module is configurable with a unique set of properties. Once generated, the module or IP appears in your designs File List.

**Process**

A process is a specific task in the overall processing of a source or project. Typical processing tasks include synthesizing, mapping, placing, and routing. You can view the available processes for a design in the Process Toolbar.

![Figure 26: Process Toolbar](image)

The process status icons are defined as follows:

- Process in initial state (not processed)
- Process completed successfully
- Process completed with unprocessed subtask(s)
- Process failed

For more detail of different designs and Export Files available, see “Process Flow” on page 19.
Task Detail View

Click Task Detail View to see detailed information of each process.

The default design flow processes are marked by check mark. To enable the remaining tasks, either check-mark the specific task and rerun the process step, or double-click the task’s name. You can also right-click on the task to show the context menu.

Once the process has finished, the process status icon next to the task replaces the gray dot.

Figure 27: Task Detail View

Processes are grouped into categories according to their functions. To learn more about each process, view “Design Flow Processes” on page 60.

Hierarchy

The Hierarchy view is a project view that displays the design hierarchy and is displayed by default. The hierarchical view is available when File List tab is selected.

Figure 28: Hierarchy View

If you would prefer that it not open by default, simply close Hierarchy View. The next time you launch the Radiant software, the Hierarchy View will not be opened. You can open it manually by selecting it from the View > Show View menu.
Right-click any of the objects in the Hierarchy View to see the available actions.

**Figure 29: Hierarchy Item Pop-up Menu**

The Hierarchy view can be selected, closed, and opened.

---

**Reports**

The Reports View provides a centralized reporting mechanism in the Tools view area. The Reports View is automatically displayed and updated when processes are run. It provides a separate tab for current implementation, enabling you to compare results quickly.

The right pane displays the report for the selected step. You can also click the icon in the toolbar.

**Figure 30: Reports View**

The Reports pane on the right shows the detail of the project summary and resource usage.

The Report View can be selected, closed, opened, detached, and attached with the Attach button. See “Basic UI Controls” on page 30.
Tool Views

The Tool view area of the UI displays the tools that are currently active. Each tool that you have opened from the toolbar or the Tools menu is displayed. The Reports and Start page, which can be opened from the toolbar or the Windows menu, are also displayed. When multiple tools are active, the display can be controlled with the tab group functions in the Window toolbar. See “Common Tasks” on page 38 for more information on tab group functions.

Each tool view is specific to its tool and can contain additional toolbars, multiple panes, or multiple windows controlled by additional tabs. See “Working with Tools and Views” on page 74 for descriptions of each tool and view, plus details on controlling their display.

The Tool views can be selected, closed, opened, detached, and attached using the Attach button. See “Basic UI Controls” on page 30

Figure 31: Tool View Tab Title

Tcl Console

The Tcl Console is an integrated console for Tcl scripting. You can enter Tcl commands in the console to control all of the functionality of the Radiant software. Use the Tcl help command (help <tool_name>*) to display a list of valid Radiant software extended Tcl commands.
Output View

The Output View is a read-only area where tool output is displayed.

Message

There are three message types available:

- Errors are displayed in red
- Warnings are displayed in orange
- General Information is displayed in blue

A red dot in the Message tab provides a visual notification that a new message/warning was received. Once the user views the notification, the dot disappears.

Right-clicking a message provides a menu of commands, including Location in > Text Editor, which opens the source file in the Text Editor and highlights the location of the problem.
Find Results

The Edit > Find in Files command enables you to search for information in the files within your project directory. The search results are then displayed in the Find Results view.

Common Tasks

The Radiant software UI controls many tools and processes. The following sections describe some of the more commonly performed tasks.

Controlling Views

All of the views in the Radiant software are controlled in a similar manner, even though the information they contain varies widely. Here are some of the most common operations:

- Open – Use the View > Show Views menu selections or right-click in the menu or toolbar areas to select a view from the pop-up menu.

- Select – If a view is already open you can select its tab to bring it to the front.

- Detach – Click the detach button in the upper right corner of the view.

- Attach – Click the attach button in the upper right corner of the view.
Move – Click and hold a view’s tab, and then drag and drop the view to a different position among the open views.

Using a Tab Group You can use the Window menu to split off a view and control it as a separate tab group. This allows you to examine two open views side by side. The controls work as follows:

- Split Tab Group – displays two views side by side. For more information, refer to Figure 36.
- Move to Another Tab Group – moves the selected tab to the other tab group.

Figure 36: Split Tab Group

Figure 37: Switch Tab Group Position
Cross-Probing Between Views

It is possible to select a data object in one view and see that same data object in a different view or views. Right-click a selected object to see if cross-probing is available. If it is, you will see a Show In sub-menu with the available views listed. If you select a view that is not yet open, the Radiant software will open it automatically. Cross-probing is available for most tool views.

Figure 38: Show In

To auto cross-probe between Floorplan View and Physical View, ensure both views are attached to the Radiant software main window and then right-click on the Floorplan View tab and select Split Tab Group.

The two views display in parallel, as shown in Figure 39.

Figure 39: Cross-Probing Views
When both Floorplan View and Physical View are open, an item that you select in one of these views is automatically selected in the other. Auto cross-probing is especially useful for immediate examination of connections in both views.

Cross-Probing of the Data Path

It is possible to view a clock or data path in timing report, then view and see that same data in different views.

During the Radiant flow, various timing analysis and reports are created. The user is able to view a specific path and it’s progression throughout various Radiant software tools. Such feature allows for flexibility and reduced debugging effort.

Next figures show a path cross-probing through the Radiant software tools, the Netlist Analyzer, Floor Plan, and Physical View.

Figure 40: Path Cross-Probing in Reports

By clicking on Netlist Analyzer icon, the user can preview the data path in the Radiant software Netlist Analyzer tool.
In some cases, the tool is unable to find the path. The user observes message:

Similarly, by clicking on FloorPlan icon, the user can easily view the same path in the Radiant software Floor Plan tool.
Same path is viewable in the Radiant software Physical View tool by clicking on Physical View icon in Reports timing report.

Figure 43: Path Cross-Probing in Physical View

Cross-probing in Encrypted Design

The Radiant software supports a path cross-probing between Netlist Analyzer, Floor Planner, and Physical View.

In the Reports tab, view any analysis report and identify a path to view. If cross-probing is available, the specific icon tools become visible, as shown in the following Figure.

Figure 44: Available tools for Path Cross-probing

Click on icon and the specific tool opens with selected path view.

Due to an encrypted design, in some cases, the path cross-probing is unable to view. The message “Cannot open encrypted design.” appears.

NOTE

Cross-probing to Netlist Analyzer is available only if selected synthesis tool is Lattice LSE.
This chapter covers projects and their elements. Implementations and strategies are explained and some common project tasks are shown.

Overview

A project is the top organizational element in the Radiant software design environment. Projects consist of design, constraint, configuration and analysis files. Only one project can be open at a time, and a single project can include multiple design structures and tool settings.

You can create, open, or import a project from the Start Page. Refer to “Getting Started” on page 10 for instructions on creating a new project.
The File List view shows a project and its main elements.

**Figure 46: Project Files in File List**

The Project menu commands enable you to do the following:

- Examine the project properties.
- Change the target device.
- Change the severity level of warning messages.
- Set the synthesis tool.
- Show the active strategy tool settings.
- Set the top level design unit.

**Figure 47: Project Menu**
Implementations

An implementation is the structure of a design and can be thought of as what is in the design. For example, one implementation might use inferred memory while another implementation uses instantiated memory. Implementations also define the constraint and analysis parameters for a project.

There can be multiple implementations in a project, but only one implementation can be active at a time. And there must be one active implementation. Every implementation has an associated active strategy. Strategies are a shared pool of resources for all implementations and are discussed in the next section. An implementation is created whenever you create a new project.

Implementations consist of the following files:

- Input files
- Pre-Synthesis constraint files
- Post-Synthesis constraint files
- Debug files
- Script files
- Analysis files
- Programming files

Adding Implementations

To add a new implementation to an existing project:

1. Right-click the project name in the File List project view

Select Add > New Implementation. In the New Implementation dialog box, you can set the implementation name, directory, default strategy, and add source files. When you select Add Source you have a choice of browsing for the source files or using a source from an existing implementation.
Figure 48: New Implementation

Notice that you have the option to “Copy source to implementation directory.” If this option is selected, the source files will be copied from the existing implementation to the new implementation, and you will be working with different source files in the two implementations. If you want the two implementations to share the same source files and stay in sync, make sure that this option is not selected.

To make an implementation active, right-click its name in the File List and choose Set as Active Implementation.

To add a file to an implementation, right-click the implementation name or any file folder in the implementation and choose Add > New File or Add > Existing File.

Cloning Implementations

To clone an implementation:

1. In File List view, right-click on the name of the implementation that you want to copy and choose Clone Implementation. The Clone Implementation dialog box opens.

2. In the dialog box, enter a name for the new implementation. This name also becomes the default name for the folder of the implementation.

3. Change the name of the implementation’s folder in the Directory text box, if desired.

4. Decide how you want to handle files that are outside of the original implementation directory. Select one of the following options:
   
   - Continue to use the existing references
     The same files will be used by both implementations.

   - Copy files into new implementation source directory
The new implementation will have its own copies that can be changed without affecting the original implementation.

5. Synthesis Tool text box specify currently selected synthesis tool. Go to Project > Active Implementation > Select Synthesis Tool to update your selection.

6. The Default Strategy text box specifies currently selected default strategy.

7. Click OK.

**Input Files**

Input files are the design source files for the project. Input files can be any combination of Verilog and VHDL.

Right-click an input file name to open a pop-up menu of possible actions for that file.

**Figure 49: Input File Actions**

You can use the “Include for” commands to specify that a source file be included for both synthesis and simulation, synthesis only, or simulation only.
Pre-Synthesis Constraint Files

Synopsys timing constraints are specified in the new .fdc file format. Legacy .sdc formats are still supported in the Radiant software and Synopsys has provided a script called sdc2fdc which does a one-time conversion of .sdc files to the new .fdc format. More information about this script can be found in the Synplify Pro release notes.

An .sdc or .fdc file can be added to an implementation if the selected synthesis tool is Synplify Pro. If the selected synthesis tool is the Lattice Synthesis Engine (LSE), a Lattice design constraint (.ldc) synthesis file can be added. Constraints in the .ldc file use the Synopsys constraint format.

An implementation can have multiple synthesis constraint files. Only one synthesis constraint file can be active at a time. Unlike Post-Synthesis constraints, a synthesis constraint file must be set as active by the user.

Post-Synthesis Constraint Files

Post-Synthesis constraint files (.pdc) contain both timing and non-timing constraint .pdc source files for storing logical timing/physical constraints. Constraints that are added using the Radiant software’s Device Constraint editor are saved to the active .pdc file. The active post-synthesis design constraint file is then used as input for post-synthesis processes.

An implementation can have multiple .pdc files, but only one can be active at a time.

Figure 50: Sample .pdc file in File List

Figure 51 shows a high-level flow of how constraints from multiple sources can be used and modified in the Radiant software.
Debug Files
The files in the Debug folder are project files for the Reveal Inserter. They are used to insert hardware debug into your design. There can be multiple debug files, and one can be set as active. To insert hardware debug into your design, right-click a debug file name and choose **Set as Active Debug File** from the pop-up menu. The debug file name becomes bold, indicating that it is active. It is not required to have an active debug file.
Script Files

The Script Files folder contains the scripts that are generated by the Simulation Wizard. After you run the Simulation Wizard, the steps are stored in a simulation project file (.spf), which can be used to control the launching of the simulator.

![Simulation Script File](image)

Analysis Files

The Analysis Files folder contains Power Calculator files (.pcf). The folder can contain multiple analysis files, and one (or none) can be set as active. The active or non-active status of an analysis file affects the behavior of the associated tool view.

Programming Files

Programming files (.xcf) are configuration scan chain files used by the Radiant Programmer for programming devices. The .xcf file contains information about each device, the data files targeted, and the operations to be performed.

An implementation can have multiple .xcf files, but only one can be active at a time. The file must be set as active by the user.
**Strategies**

Strategies are collections of all the implementation-related tool settings in one convenient location. Strategies can be thought of as recipes for how the design will be implemented. An implementation defines *what* is in the design, and a strategy defines *how* that design will be run. There can be many strategies, but only one can be active at a time. There must be one active strategy for each implementation.

The Radiant software provides three predefined strategies: Area, Quick and Timing. It also enables you to create customized strategies. Predefined strategies cannot be edited, but they can be cloned, modified, and saved as customized user strategies. Customized user strategies can be edited, cloned, and removed. All strategies are available to all of the implementations, and any strategy can be set as the active one for an implementation.

To create a new strategy from scratch, choose **File > New > Strategy**. In the New Strategy dialog box, enter a name for the new strategy. Specify a file name for the new strategy and choose a directory to save the strategy file (.sty).

The new strategy is with all the default settings of the current design. You can modify its settings in the Strategies dialog box.

If you want to save the strategy changes to your current project, choose **File > Save Project** from the Radiant software main window.

**Figure 54: Creating a New Strategy from Scratch**

![New Strategy dialog box]

To create a new strategy from an existing one, right-click the existing strategy and choose **Clone <strategy name> Strategy**. Set the new strategy’s ID and file name.
Figure 55: Cloning to Create a New Strategy

To make a strategy active, right-click the strategy name and choose **Set as Active Strategy**. To change the settings in a strategy:

1. Double-click the strategy name in the File List view
2. Select the option type to modify
3. Double-click the Value of the option to be changed

The default values are displayed in plain blue text. Modified values are displayed in italic bold text.

Strategies are design data independent and can be exported and used in multiple projects.

**Area**

The Area strategy is a predefined strategy for area optimization. Its purpose is to minimize the total logic gates used while enabling the tight packing option available in Map.

Applying this strategy to large and dense designs might cause difficulties in the place and route process, such as longer time or incomplete routing.
Quick

The Quick strategy is a predefined strategy for doing an initial quick run. This strategy uses a very low effort level in placement and routing to get results with minimum run time. If your design is small and your target frequencies are low, this is a good strategy to try. Even if your design is large, you might want to start with this strategy to get a first look at place-and-route results and to tune your constraint file with minimum runtime.

The Quick strategy will give you results in the least possible time. However, the quality of these results in terms of achieved frequency will probably be low, and large or dense designs might not complete routing.
Timing

The Timing strategy is a predefined strategy for timing optimization. Its purpose is to achieve timing closure. The Timing strategy uses very high effort level in placement and routing. Use this strategy if you are trying to reach the maximum frequency on your design. If you cannot meet your timing requirements with this strategy, you can clone it and create a customized strategy with refined settings for your design. This strategy might increase your place-and-route run time compared to the Quick and Area strategies.
User-Defined

You can define your own customized strategy by cloning and modifying any existing strategy. You can start from either a predefined or a customized strategy.

Common Tasks

Working with projects includes many tasks, including: creating the project, editing design files, modifying tool settings, trying different implementations and strategies, and saving your data.

Creating a Project


Changing the Target Device

There are two ways to access the Device Selector dialog box for changing the target device:
Double-click the device in the project File List view or right-click it and choose Edit.

Choose Project > Device.

**Setting the Top Level of the Design**

If multiple top levels exist in the hierarchy of your HDL source files, you will need to set the top-level design unit. After generating the hierarchy, choose Project > Active Implementation > Set Top-Level Unit. Alternatively, right-click the implementation and choose this command from the pop-up menu.

![Figure 59: Top-Level Design Unit](image)

In the Project Properties dialog box, select Value next to Top-Level Unit and select the desired top level from the list.

You can also use the Hierarchy View to set the top-level. Right-click a level you want to be the top-level in the Hierarchy View and choose Set Top-Level Unit.

**Editing Files**

You can open any of the files for editing by double-clicking or by right-clicking and choosing Open or Open with.
Saving Project Data
In the File menu are the following selections for saving your design and project data:

- Save – saves the currently active item.
- Save As – saves the active item using a different file name.
- Save All – saves all changed documents.
- Save Project – saves the current project.
- Save Project As – saves the active project using a different project name.
- Archive Project – creates a zip file of the current project in a location you specify.
Chapter 6

Radiant Software Design Flow

This chapter describes the design flow in the Radiant software. Running processes and controlling the flow for alternate what-if scenarios are explained.

Overview

The FPGA implementation design flow in the Radiant software provides extensive what-if analysis capabilities for your design. The design flow is displayed in the Task Detail View at the right end of the Process Toolbar.

Figure 60: Design Flow Shown in Task Detail View

1. Double click a task to run the flow
2. Right click to show context menu
Design Flow Processes

The design flow is organized into discrete processes, where each step allows you to focus on a different aspect of the FPGA implementation.

**Synthesize Design**  This process runs the selected synthesis tool (Lattice Synthesis Engine is the default) in batch mode to synthesize your HDL design.

- Synthesis Tool - identifies the selected synthesis tool, Lattice Synthesis Engine or Synplify Pro.
- Post-Synthesis Timing Analysis - generates timing analysis files.
- Post-Synthesis Simulation File - generates a post-synthesis netlist file `<file_name>_syn.vo` used for Post-Synthesis Simulation.

**Map Design** This process maps the design to the target FPGA and produces a mapped Unified Database (.udb) design file. Map Design converts a design’s logical components into placeable components.

- Map Timing Analysis - generates timing analysis files.

**Place & Route Design** This process takes mapped physical design files and places and routes the design. The output can be processed by the design implementation tools. Timing analysis files can also be generated.

- Place & Route Timing Analysis - generates timing analysis files.
- I/O Timing Analysis - generates I/O timing analysis files.

**Export Files** This process generates the IBIS, simulation, and programming files that you have selected for export:

- Bitstream File – generates a configuration bitstream (bit images) file, which contains all of the design’s configuration information that defines the internal logic and interconnections of the FPGA, as well as device-specific information from other files.
- IBIS Model – generates a design-specific I/O Buffer Information Specification model file (.ibs). IBIS models provide a standardized way of representing the electrical characteristics of a digital IC’s pins (input, output, and I/O buffers).
- Gate-Level Simulation File – generates a Verilog netlist of the routed design that is back annotated with timing information. This generated .vo file enables you to run a timing simulation of your design.

The files for export can also be generated separately by double-clicking each one.

Running Processes

You can perform the following actions for each step in the process flow:

- Run – runs the process, if it has not yet been run.
- Force Run – reruns a process that has already been run.
The state of each process step is indicated with an icon to the left of the process. The process status icons description is described in “Process” on page 33.

The Reports View displays detailed information about the process results, including the last process run. The Messages section shows warning and error messages and allows you to filter the types of messages that are displayed. See “Reports” on page 35.

Figure 61: Reports View of Last Process Run

IP Encryption Flow

IP Encryption Flow

IP encryption flow enables you to protect your IP design. Following the industry standard, the Radiant software, through the IP encryption flow, allows the partnership between the IP Vendor, supported EDA vendor, and Lattice.
The encryption flow uses symmetric and asymmetric encryption methods to maximize the design security. The symmetric method only involves a single symmetric key for both, encryption and decryption. The asymmetric method involves the public-private key pair. The public key is published by a vendor and is used by the Radiant software. The private key is never revealed to the public.

The Radiant software supports these cryptographic algorithms:

- AES-128/AES-256: symmetric algorithm used to encrypt the content of the HDL source file.
- RSA-2048: asymmetric algorithm used to obfuscate a key used in HDL file encryption. The RSA is defined by the public-private key pair. You must be familiar with both keys in order to perform RSA decryption.

**HDL File Encryption Flow**

The current software version supports encryption of a single HDL source file per a single command.

The overall HDL file encryption flow is summarized in these steps:

- The source file of the IP design is AES encrypted using a symmetric Session key. The AES encryption uses CBC-128 or CBC-256 algorithm. In the source files, this section is referred to as a data block.
- The Session key is RSA encrypted using the vendor’s Public Key. In the source files, this section is referred to as a key block. Multiple key blocks may be present in the source file.
The encrypted Session key and the encrypted design are merged to file generally named the Encrypted RTL.

Each encrypted source file contains a single data block and one or more key blocks. The number of key blocks depends on the number of provided vendor’s public key.

**NOTE**

To decrypt an encrypted source file, you must perform the IP encryption flow steps in the reverse order.

During the next step in the design flow, typically a synthesis, the Encrypted RTL is decrypted to access the original IP design, as shown in the following figure.

**Figure 63: HDL Encryption Flow**

By separating the encryption of data and key, you can use public keys from different vendors to encrypt same HDL file.

For more information on how to perform HDL encryption, refer to “Running HDL Encryption from the Command Line” on page 109.

**HDL File Encryption Steps**

This section provides step-by-step instructions on how to encrypt an HDL file.

The Radiant software provides the key templates you can simply drag-and-drop into an HDL file. Each key template is specific to an EDA vendor providing the value of a public key.

To view the templates in Project Navigator, choose **Source Template > Verilog > Encryption Templates** and select the EDA specific key template.

Currently, the Radiant software supports these encryption templates:

- Lattice Semiconductor
- Synplicity-1
Step 1: Prepare the HDL file

Annotate the HDL source file with protected pragmas. Protected pragmas provide information regarding type of the key used to encrypt the HDL file, the name of the key, and the encryption algorithm.

In this example, HDL source file will be encrypted by the Lattice Public Key.

```vhdl
'pragma protect version=1
'pragma protect encoding="enctype="base64"

// optional information
'pragma protect author="<Your_Name>"
'pragma protect author_info="<Your_Information>"
```

Step 2: Specify the portion of HDL source file that shall be encrypted

Annotate the HDL file to specify the encryption. Only the portion defined within these protected pragmas is encrypted.

```vhdl
'pragma protect begin
// HDL portion that shall be encrypted
'pragma protect end
```

Step 3: Prepare Key

Define the key with which the HDL file should be encrypted. Each key definition must contain the following information:

- key_keyowner: specify the owner of the key
- key_keyname: specify the name of the key. Same owner may provide multiple keys.
- key_keymethod: specify the used cryptographic algorithm. Current version supports RSA algorithm.
- key_public_key: specify the exact value of the key.

The key definition can be done in two ways:

**Defining the key in the key.txt file:** The public encryption key or keys can be defined in any .txt file. The key file may contain a single public key or a list of all available public keys. In the Radiant software, all common EDA vendor public keys are located in `<Radiant_installed_directory>/ispfpga/data/key.txt` file.
The following is an example of Lattice Public Key defined in key.txt file:

```
'pragma protect key_keyowner= "Lattice Semiconductor"
'pragma protect key_keyname= "LSCC_RADIANT_2"
'pragma protect key_method="rsa"
'pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vScs70hQJiNAWJR5unW/Opw/LF171eA1s3b0YrE0k1dxbai+nd1eo8xFz2tbdxUzuR6Srve
xrSsJ9b9bQ1o2u8jOz3X7AmRvlwK88708Dq6LDZ4M33hUKfDDwkp2Eavsf
LjEzVeCvXq7f/ybDeuN8xRQCSKJ7aguG6ko1f6Ro02z11jzDLQ2zm2qYF888pU0
etD8/uv53w0f8rH3MB0B+xnc2imvSldgNWhIc6cxZ1x5CD4y8inCbcLy/0Qrf6
sdTN5Ag20ZhjeNdzmSqWqL2JTDw+0u2fWzhEd0I/HN0y4NMre9fNn8nqXryE7
1wIDAQAB
```

Defining the key directly in the HDL file: You may define the Public Key directly in the HDL file. You may define one or more keys.

```
'pragma protect key_keyowner= "Lattice Semiconductor"
'pragma protect key_keyname= "LSCC_RADIANT_2"
'pragma protect key_method="rsa"
'pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vScs70hQJiNAWJR5unW/Opw/LF171eA1s3b0YrE0k1dxbai+nd1eo8xFz2tbdxUzuR6Srve
xrSsJ9b9bQ1o2u8jOz3X7AmRvlwK88708Dq6LDZ4M33hUKfDDwkp2Eavsf
LjEzVeCvXq7f/ybDeuN8xRQCSKJ7aguG6ko1f6Ro02z11jzDLQ2zm2qYF888pU0
etD8/uv53w0f8rH3MB0B+xnc2imvSldgNWhIc6cxZ1x5CD4y8inCbcLy/0Qrf6
sdTN5Ag20ZhjeNdzmSqWqL2JTDw+0u2fWzhEd0I/HN0y4NMre9fNn8nqXryE7
1wIDAQAB
```

If the key is defined directly in the HDL file, you don’t need to provide `-k` option in `encrypt_hdl` command.

**Note**

The key defined directly in HDL source file has preference over the key defined in the key.txt file.

**Step 4: Select the encryption algorithm for data encryption**

The Radiant software supports both, a 128-bit and a 256-bit advanced encryption standard (AES) with CBC mode. Select the type of algorithm by defining one of the two options. The default is set to 256-bit AES with CBC mode.

```
'pragma protect data_method="aes128-cbc"
```

or

```
'pragma protect data_method="aes256-cbc"
```

**Step 5: Run ‘encrypt_hdl’ Tcl Command**

In the Tcl console window, type in the command to encrypt an HDL file. The option `-k` may or may not be used depending the location of the key file. The language selection `-l` and creation of new output file `-o` are optional; selects Verilog by default. If you don’t specify the output file, the tool generates a new output file named `<input_file_name>_enc.v`.

---

Lattice Radiant Software 2.0 User Guide 65
If key was defined in the key.txt file: The command will encrypt the HDL file with all keys defined in the key.txt file.

```
encrypt_hdl -k <keyfile> -l <verilog | vhdl> -o <output_file> <input_file>
```

If key was defined directly in the HDL file:

```
encrypt_hdl -l <verilog | vhdl> -o <output_file> <input_file>
```

The encrypted file is located at the path specified in the `encrypt_hdl` command.

**Step 6: Activate the encrypted HDL source file in Project File**

In the Radiant software File List, add the generated file into the project. Right-click on the encrypted file and select **Include in Implementation**.

To view example of Verilog or VHDL pragma annotated HDL source file, visit “Defining Pragmas” on page 111.

**Implementation Flow and Tasks**

Implementations organize the structure of your design and allow you to try alternate structures and tool settings to determine which ones will give you the best results.

To help determine which scenario best meets your project goals, use a different implementation of a design using the same tool strategy, or run the same implementation with different strategies. Each implementation has an associated active strategy, and when you create a new implementation you must select its active strategy.

To try a new implementation with different strategies, you must create a new implementation/strategy combination.

1. Right-click the project name in the File List.
2. Choose **Add > New Implementation**.
3. Select a source from an existing implementation using the Add Source drop down menu.
4. Choose a currently defined strategy using the Default Strategy drop down menu.
To use the same source for new and existing implementations, make sure that the “Copy source to implementation directory” option is not selected. This will ensure that your source is kept in sync between the two implementations.

**Synthesis Constraint Creation**

Synthesis constraints can be added to a design implementation in the format of the Synopsys® Design Constraint language, while constraints can be added in the Synopsys standard timing constraints format in the form of FPGA Design Constraint (LDC, PDC, FDC format) files.

If you are using the Lattice Synthesis Engine, the synthesis constraints will be included in an .ldc file. If you are using Synplify Pro for synthesis, the constraints will be included in an .fdc file. The older .sdc file format is also supported for constraints.

To create a new synthesis constraint file, right-click the Synthesis Constraint Files folder in the File List pane and choose **Add > New File**. In the New File dialog box, select one of the following and give the file a name:

- Pre-Synthesis Constraint Files (.ldc)
- Post-Synthesis Constraint Files (.pdc)
- Synplify Pro Constraint Files (.fdc)
The .ldc, .pdc, or .fdc file will open in the Source Editor to allow you to manually add the constraints. You can use the Pre-Synthesis Timing Constraint Editor tool to add pre-synthesis timing constraints to .ldc and the Post-Synthesis Timing Constraint Editor tool to add logic view level post-synthesis timing constraints to .pdc files. You can also use the Device Constraint Editor and Floorplan View to add physical constraints to .pdc files. For detailed information about setting constraints, see Applying Design Constraints and the Constraints Reference Guide in the Radiant software online Help.

An alternative way of adding constraint files is through a Source Template. To view a constraint template, click on the Source Template tab on the left-hand side of the Project Navigator pane. If not selected, make sure it is enabled in View > Show Views > Source Template. The list of constraint templates includes the timing constraints, physical constraints, and user templates. Select specific template and copy paste into your active design.

### Constraint Creation

LDC (pre-synthesis) and PDC (post-synthesis) files are used to input timing and physical constraints. The following steps illustrate how to assign and edit constraints in the Radiant software and implement them at each stage of the design flow.

1. If desired, define some constraints at the HDL level using HDL attributes. These source file attributes are included in the Unified Database (UDB), and will be displayed in the Radiant software after the Map Design process is run. The following is an example of applying the LOC attribute in Verilog source code:

```verilog
module top (
    input clk1,
```
input datain /* synthesis loc = B12 */,
output ff_clk1out
)

For more information on HDL Attributes, see the topic “HDL Attributes” in the Constraints Reference Guide section of the Radiant software online Help.

2. Open one or more of the following tools to create new constraints or to modify existing constraints from the source files.

   ▶ Device Constraint Editor, which consists of:
     ▶ Spreadsheet View -- This is the primary view for setting constraints.
     ▶ Package View -- Examines the pin layout of the design; modifies signal assignments and reserve pin sites that should be excluded from placement and routing; runs PIO design rule check to verify legal placement of signals to pins.
     ▶ Device View -- Examines FPGA device resources.
     ▶ Netlist View - Shows Port types (Input, Output) and Groups

   ▶ Timing Constraint Editor. Timing/Physical constraints are entered through:
     ▶ Pre-Synthesis Timing Constraint Editor - Used to enter pre-synthesis timing constraints such as clocks, clock latency/uncertainty/Group, Input/Output delays, timing exceptions and attributes.
     ▶ Post-Synthesis Timing Constraint Editor - This post-synthesis version of the Timing Constraint Editor is used to enter logic view level timing constraints and physical constraints.
     ▶ Floorplan View – Examines the device layout of the design; draws bounding boxes for GROUPs, draws REGIONs for the assignment of groups or to reserve an area, reserves sites and REGIONs that should be excluded from placement.

3. Save the constraints to the pre-synthesis constraint file (.ldc) or post-synthesis constraint file (.pdc).

4. Run the Map Design process (Map).

5. Run the Map Timing Analysis process and examine the timing analysis report. This is an optional step, but it can be a quick and useful way to identify serious timing issues in design and/or constraint errors (syntax and semantic). Modify constraints as needed and save them.

6. Run the Place & Route Design process.

7. Open views directly or by cross-probing to examine timing and placement and create new GROUPs. Also examine the Place & Route Timing report.

   ▶ Timing Analysis View – Examine details of timing paths and cross probes selected paths to Floorplan and Physical Views.

8. Modify constraints or create new ones using appropriate constraint tool’s views. Save the constraints changes and rerun the Place & Route Design process.
Simulation Flow

The simulation flow in the Radiant software supports source files that can be set in the File List view to be used for the following purposes:

- Simulation & Synthesis (default)
- Simulation only
- Synthesis only

This allows the use of test benches, including multiple file test benches. Additionally, multiple representations of the same module can be supported, such as one for simulation only and one for synthesis only.

The user can add top level signals to the waveform display in the simulator and to automatically start the simulator running.

The Simulation Wizard automatically includes any files that have been set for simulation only or for both simulation and synthesis. The user can select the top of the design for simulation independent of the implementation design top. This allows easy support for test bench files, which are normally at the top of the design for simulation but not included for implementation. The implementation wizard exports the design top to the simulator, along with source files, and set the correct top for the .spf file if running timing simulation.

The File List view shows an implementation's input files for simulation. This is a listing of source files and does not show design hierarchy.

Figure 66: Input Files for Simulation
After you add a module, use the **Include for** menu to specify how the module file is to be used in the design.

**Figure 67: ‘Include For’ Input Files**

---

**Simulation Wizard Flow**

When you are ready to simulate, export the design using the Simulation Wizard. Aside of the RTL simulation, you can perform the Post-Synthesis simulation and Post-Place & Route back annotated netlist simulation.

Post-Synthesis Simulation File generates a Verilog netlist of post-synthesis netlist (_syn.vo) file. Similarly, the Gate-Level Simulation File generates a Verilog netlist of the routed design (.vo file) that is back annotated with timing information (.sdf file). The generated file enables you to run a timing simulation of your design. For more details on how to generate these files, see “Task Detail View” on page 34.

Choose **Tools > Simulation Wizard** or click the Simulation Wizard icon on the toolbar. The wizard leads you through a series of steps:

1. Select a simulation project name and location.
2. Specify the simulator to use (if you have more than one installed).
3. Select the process stage to use (RTL, Post-Synthesis, or Post-Route Gate-Level + Timing).
4. Select the language (Verilog) and source files.

You can also run the simulation directly from the wizard, as shown in Figure 68.
Figure 68: New Simulation Project

After you have set up the simulator project and specified the implementation stage and source files to be included, the Simulation Wizard parses the HDL and test bench. The last step is to specify the simulation top module.

Figure 69: Simulation Top Module

In some designs, the compile order of the HDL files passed to the simulator might result in compilation warnings. In most cases, these compilation warnings can be safely ignored. The warnings can be eliminated in one of two ways:

- The correct compilation order for the HDL files can be set manually in the File List view. After the correct order is determined, the files are sent to the simulator, which will eliminate any compilation warnings.
The correct compilation order for the HDL files can be set in the Simulation Wizard during the “Add and Reorder” step. After the correct order for the files is set manually, the files will be sent to the simulator, which will eliminate any compilation warnings.
Chapter 7

Working with Tools and Views

This chapter covers the tools and views controlled from the Radiant software framework. Tool descriptions are included and common tasks are described.

Overview

The Radiant software design environment streamlines the implementation process for FPGAs by combining the tool control and data views into one common location. Two main features of this design environment make it easy to keep track of unsaved changes in your design and examine data objects in different view.

Shared Memory

The Radiant software uses shared memory that is accessed by all tools and views. As soon as design data has been changed, an asterisk * appears in the tab title of the open views, notifying you that unsaved changes are in memory.

Cross Probing

Shared design data in the Radiant software enables you to select a data object in one view and display it in another. This cross-probing capability is especially useful for displaying the physical location of a component or net after it has been implemented. A user can click on a hyper-link icon to cross probe into the specific tool. The Radiant software supports:

- Post-Synthesis timing report links to Netlist Analyzer.
- Map & PAR timing reports link to Floorplan view and Physical view.
View Menu Highlights

The View menu and toolbar control the display of all toolbars, project views and display control. Also included in the View menu are the important project-level features: Start Page, and Reports.

Start Page

The Start Page is displayed by default when you run the Radiant software. The Start Page enable you to open projects, read product documentation, and view the software version and updates. You can modify startup behavior by choosing Tools > Options > General > Startup.

Figure 70: Default Start Page

The Start Page gives you quick access to recent projects and to product documentation.

Reports

The Reports view provides one central location for all project and tool report information. It is displayed by default when a project is open. Alternatively, click on icon in toolbar.
The Reports view is organized into Project Summary, Synthesis Reports, Map Reports, Place & Route Reports, Export Reports, and Misc Reports.

The different file icons indicate:
- A report has been completed (blue check mark).
- A report has never been generated (gray circle).
- A report is out of date (orange question mark).

Select any item to view its report.

The Reports view also supports a path cross-probing through the timing or analysis reports. The user is able to view a specific clock or data path in Radiant software tools such as Netlist Analyzer, Floor Planner, or Physical View, by clicking the icon next to the path.
To learn more about cross-probing, view “Cross-Probing Between Views” on page 40.

## Tools

The entire FPGA implementation process tool set is contained in the Radiant software. You can run a tool by selecting it from the Tools menu or toolbar.

This section provides an overview of each of these tools. More detailed information is available in their respective user guides, which you can access from the Start Page or from the Radiant software Help. Detailed descriptions of external tools can be found in their product documentation as well.

If you are viewing an encrypted design, some secured objects may not be visible in the selected tool. To learn more, view “Secure Objects in the Design” section of Online Help.

### Timing Constraint Editor

The Timing Constraint Editor is used to edit pre-synthesis timing (.ldc) constraints and post-synthesis constraints stored in a .pdc file. The Timing Constraint Editor consists of two tools:

- Pre-Synthesis Timing Constraint Editor
- Post-Synthesis Timing Constraint Editor
Both tools have identical interfaces and the entry mechanisms of the constraints are also the same. The key differences are that pre-synthesis constraints are entered pre-synthesis and are synthesized by the chosen synthesis tool. The post-synthesis constraints are already synthesized and populated in each of the different constraints tabs. The user cannot modify the post-synthesis constraints that were populated from pre-synthesis, but can supply new constraints (physical and timing) to either override the existing one already populated or supply brand new constraints to better constrain the design for improved performance upon analysis.

The different constraint types are entered through these tabs:

- **Clock** - used to define the clocking scheme of the design.
- **Generated Clock** - used to define generated clocks such as from PLLs.
- **Clock Latency** - is the delay between the clock source and clock pin. Used to define the latency in terms of Rise and Fall times.
- **Clock Uncertainty** - is the jitter difference two signals possibly caused by clocks. Used to define the amount of uncertainty of a clock or during clock domain transfer.
- **Clock Group** - used to specify which clocks are not related in terms of being Logically / Physically Exclusive and Asynchronous.
- **Input/Output Delay** - used for setting Input and Output delays.
- **Timing Exception** - used to set Min / Max, False and Multicycle path constraints.
- **Attribute** - used to set synthesis attributes.

**Figure 73: Pre-Synthesis Timing Constraint Editor**
Constraint Propagation

Constraint propagation is not a standalone tool, nor does the user have to provide any user input to take advantage of this feature.

Users will usually define constraints for their top level design as well as include other constraint files for any customer included IP or IPs that are generated via Radiant tools such as IPExpress. Constraints defined at the module IP level may not contain the correct hierarchical names and hence will not be applied correctly when synthesized. To help the user honor as much of their supplied constraints as possible, this feature will run on all the input constraints a DRC that will write out a new constraint file to support hierarchical constraints such as soft-IP constraint and honor top level constraints.

For more information about Constraint Propagation, refer to Constraint Propagation in the Radiant software online Help.

Device Constraint Editor

The Device Constraint Editor is used to edit post-synthesis constraints. These constraint editing views are available from the Radiant software toolbar and Tools menu.

All modified constraints are saved to a .pdc file and the flow returned to Map. The Device Constraint Editor shows the pin layout of the device and displays the assignments of signals to device pins. This view allows the user to edit these assignments, and reserve sites on the layout to exclude from placement and routing. The Device Constraint Editor is also the entry mechanism for physical constraints.

Device Constraint Editor views, shown in Figure 74, enable you to develop constraints that will shorten turn-around time and achieve a design that conforms to critical circuit performance requirements.
SSO Analysis
Radiant enables you to run an analysis of simultaneous switching outputs (SSO). SSO analysis describes the noise on signals caused by a large number of out drivers that are switching at the same time. Analysis of SSO helps ensure that your I/O standards and power integrity requirements of the PCB design. This tool can be accessed in the Device Constraint Editor in the SSO tab at the bottom. From there a user can set output loads, Ground plane PCB noise, SSO Allowance % and Power plane PCB noise values. Once the flow is re-run, a SSO report will be generated.

Netlist Analyzer
Netlist Analyzer works with Lattice Synthesis Engine (LSE) to produce schematic views of your design while it is being implemented. (Synplify Pro also provides schematic views.) Use the schematic views to better understand the hierarchy of the design and how the design is being implemented. The Netlist Analyzer window has four parts, as shown in Figure 75.
Figure 75: Netlist Analyzer

Tool bar provides buttons for various functions.

Browser provides nested lists of module instances, ports, and nets.

Schematic view shows a schematic of the design. Depending on the size of the design, the schematic may be made of multiple sheets.

World View, which is a miniature view of the sheet, helps you pan and zoom in the schematic view.

Netlist Analyzer can have multiple schematics open. The open schematics are shown on tabs along the bottom of the window.

There are several ways to adjust the view of a schematic and to navigate through the hierarchy. For more information on how to do this, in addition to detailed information about Netlist Analyzer, see About Netlist Analyzer in the Radiant software online Help.

Floorplan View

Floorplan View provides a large-component layout of your design, and is available as soon as the target device has been specified. Floorplan View displays user constraints, placement and routing information. All connections are displayed as fly-lines.

Floorplan View allows you to create REGIONs and bounding boxes for GROUPs, in addition to specifying the types of components and connections to be displayed. As you move your mouse pointer over the floorplan layout, details are displayed in tool-tips and in the status bar, including:

- Number of resources for each GROUP and REGION.
- Number of utilized slices for each PLC component.
- Name and location of each component, port, net, and site.
New to the floorplan view are the *Congestion Timing Map View* allowing you to debug timing congested areas of your design. This view gives the timing of the most critical paths within the slack threshold input. Furthermore, a new *Congestion View* is a read-only view that shows the most congested regions based on wire length and pins selected.
Figure 78: Congestion View

Figure 79: IO Planner View

IO Planner View
Stemming from the Floorplan view is now a IO Planner view. Floorplan view is for GROUPS and REGIONS assignment. IO Planner is used for IO assignment such as DDR Interface, DQS and clock assignments. IO resource utilization can be displayed by hovering over and displayed in tool tips.

Physical View
Physical View provides a read-only detailed layout of your design that includes physical wire connections. Routed connections are displayed as Manhattan-style lines, and unrouted connections are displayed as fly-lines.
As you move your mouse pointer slowly over the layout, the name and location of each REGION, group, component, port, net, and site are displayed as tool tips and also appear in the status bar. The tool tips and status bar also display the group name for components that are members of a group.

The Physical View toolbar allows you to select the types of elements that will be displayed on the layout, including components, empty sites, pin wires, routes, and timing paths. Physical View is available after placement and routing.

**Timing Analysis View**

Timing Analysis View analyzes timing constraints that are present in the .ldc and .pdc files. These timing constraints are defined in the Timing Constraint Editor or in a text editor before the design is mapped. A Timing Analysis report file, which shows the results of timing constraints, is generated each time you run the LSE, Map Timing Analysis process or the Place & Route Timing Analysis (PAR) process. Place & Routing Timing Analysis results can then be viewed in the Timing Analysis View windows.

The Map Timing Analysis report (.tw1) contains estimated routing that can be used to verify the expected paths and to provide an estimate of the delays before you run Place & Route. The PAR Timing Analysis report (.twr) contains delays based on the actual placement and routing and is a more realistic estimate of the actual timing.
Figure 81: Timing Analysis View

Timing Analysis View consists of five tabs of information:

- General Information
- Paths for All the Timing Constraints
- Critical Endpoint Summary
- Unconstrained Endpoint Summary
- Query

Each tab can be detached from the main window, rearranged, and resized. When you select a constraint from the Constraint pane, you can view the path table details on one pane, and Timing Analysis View report in the other. For detailed information about Timing Analysis View see Analyzing Static Timing in the Radiant software online Help.

Reveal Inserter

Reveal Inserter lets you add debug information to your design to allow hardware debugging using Reveal Analyzer. Reveal Inserter enables you to select the design signals to use for tracing, triggering, and clocking. Reveal Inserter will automatically generate the debug core, and insert it into a modified design with the necessary debug connections and signals. Reveal Inserter supports VHDL and Verilog sources. After the design has been modified for debug, it is mapped, placed and routed with the normal design flow in the Radiant software.
For more information about Reveal Inserter, refer to the *Reveal User Guide for Radiant Software*. Also, refer to *Testing and Debugging On-Chip* in the Radiant software online Help.

**Reveal Analyzer**

After you generate the bitstream, you can use Reveal Analyzer to debug your FPGA circuitry. Reveal Analyzer gives you access to internal nodes inside the device so that you can observe their behavior. It enables you to set and change various values and combinations of trigger signals. After the specified trigger condition is reached, the data values of the trace signals are saved in the trace buffer. After the data is captured, it is transferred from the FPGA through the JTAG ports to the PC.

For more information about Reveal Analyzer, refer to the *Reveal User Guide for Radiant Software*. Also, refer to *Testing and Debugging On-Chip* in the Radiant software online Help.

**Reveal Controller**

Reveal Controller allows you to emulate an otherwise unavailable environment for power debug. For example, your evaluation board would only have a limited number of LEDs or switches but the virtual environment enables up to 32 bits. Register memory mapping and dumping of values is also easily manifested while visibility into Hard IPs is also enabled.

For more information about Reveal Controller, refer to the *Reveal User Guide for Radiant Software*. Also, refer to *Testing and Debugging On-Chip* in the Radiant software online Help.

**Power Calculator**

Power Calculator estimates the power dissipation for your design. It uses parameters such as voltage, temperature, process variations, air flow, heat sink, resource utilization, activity and frequency to calculate the device power consumption. It reports both static and dynamic power consumption.

Power Calculator files (.pcf) are managed in the Analysis Files folder of the File List.
To launch Power Calculator from the Radiant software, choose **Tools > Power Calculator** or click the Power Calculator button on the toolbar.

When Power Calculator is launched, the .pcf file it uses depends on the following conditions:

- If an active .pcf file exists, it will be used.
- If an active or inactive .pcf file in the File List Analysis Files folder is double-clicked, it will be used.
- If no .pcf file exists, Power Calculator will perform power calculations based on the current open design.

**Power Calculation Modes**  
Power Calculator opens in estimation mode or calculation mode, depending on the status of the selected .pcf file. If it opens in calculation mode, the Bank settings will be from background power database not from the constraint file. When you make certain design changes in calculation mode, Power Calculator reverts to estimation mode.

**Power Calculator Pages**  
When Power Calculator opens, it displays the Power Summary page, which enables you to change the targeted device, operating conditions, voltage, and other basic parameters. Updated estimates of power consumption are then displayed based on these changes. Tabs for other pages, including Power Matrix, Logic Block, Clocks, I/O, I/O Term, Block RAM, Graph, and Report, are arranged across the bottom. The number and types of these pages depends on the target device.
Non-Integrated Power Calculator

Power Calculator is also available as a non-integrated tool, which you can launch without opening the Radiant software. The non-integrated Power Calculator provides all the same functionality as the integrated version. To open the non-integrated Power Calculator from the Windows Start menu, select **All Programs > Lattice Radiant Software > Accessories > Power Calculator**. The Startup Wizard enables you to create a new Power Calculator project, based on a selected device or a processed design, or to open an existing Power Calculator project file (.pcf).

For more information on Power Calculator see **Analyzing Power Consumption** in the Radiant software online Help.

ECO Editor

The Engineering Change Order (ECO) Editor enables you to safely make changes to an implemented design without having to rerun the entire process flow. Choose **Tools > ECO Editor** or click the ECO Editor button 🔖 on the toolbar.
ECOs are requests for small changes to be made to your design after it has been placed and routed. The changes are directly written into the Unified Database (.udb) file without requiring that you go through the entire design implementation process.

ECOs are usually intended to correct errors found in the hardware model during debugging. They are also used to facilitate changes that had to be made to the design specification because of problems encountered when other FPGAs or components of the PC board design were integrated.

The ECO Editor includes windows for editing I/O preferences, and memory initialization values. It also provides a Change Log window that enables you to track the changes between the modified .udb file and the post-PAR .udb file.

**Note**

After you edit your post-PAR UDB file, your functional simulation and timing simulation will no longer match.

For more information, see *Applying Engineering Change Orders* in the Lattice Radiant software online Help.

**Programmer**

The Radiant software Programmer is a system for programming devices. The software supports serial programming of Lattice devices using PC and Linux environments. The tool also supports embedded microprocessor programming.
For more information about Programmer and related tools, refer to the Programming Tools User Guide for Radiant Software. Also, refer to Programming the FPGA in the Radiant software or Stand-Alone Programmer online Help.

Run Manager

Run Manager runs the processes for the different implementation/strategy combinations. Choose Tools > Run Manager or click the Run Manager button on the toolbar.

Run Manager takes the design through the entire process flow for each selected implementation. If you are running on a multi-core system, Run Manager will distribute the iterations so that they are executed in parallel. The options “Maximum number of implementation processes in run manager” and “Maximum number of multi-par processes in run manager” are available in the General section of the Options dialog box. Choose Tools > Options to access it. These options enable you to set the maximum number of processes to run in parallel. Generally, the maximum number of processes should be the same as the number of cores in your processor; but if the strategy is using the “Multi-Tasking Node List” option for Place & Route Design, this number should be set to one.

You can use the Run Manager list to set an implementation as active. Right-click the implementation/strategy pair and choose Set as Active.

For an implementation that uses multiple iterations of place-and-route, you can select the iteration that you want to use as the active netlist for further processes. Expand the implementation list, right-click the desired iteration, and choose Set as Active. The active iteration is displayed in italics.

To examine the reports from each process, set an implementation as active, and then select the Reports View.

See the “Managing Projects” section of the Radiant software online Help for more information about using implementations, strategies, and Run Manager.

Synplify Pro for Lattice

Synplify Pro for Lattice is an OEM synthesis tool used in the Radiant software design flow. Synplify Pro runs in batch mode when you run the Synthesize Design step in Process View. To examine the output report, select Synplify Pro in the Process Reports folder of Reports View.
You can also run Synplify Pro in interactive mode. Choose **Tools > Synplify Pro for Lattice** or click the Synplify Pro button on the toolbar.

For more information, see the *Synplify Pro User Guide*, which is available from the Radiant software Start Page or the Synplify Pro Help menu.

### Active-HDL Lattice Edition

The Active-HDL Lattice Edition tool is an OEM simulation tool that is closely linked to the Radiant software environment. It is not run as part of the Process implementation flow. To run Active-HDL, choose **Tools > Active-HDL Lattice Edition** or click the Active-HDL button on the toolbar.

See “Simulation Flow” on page 70 for more information about simulating your design. See “Simulation Wizard Flow” on page 71 for information about creating a simulation project to run in Active-HDL.

For complete information about Active-HDL, see the *Active-HDL Online Documentation*, which is available from the Radiant software Start Page or the Active-HDL Help menu.
Simulation Wizard

The Simulation Wizard enables you to create a simulation project for your design. To open Simulation Wizard, choose Tools > Simulation Wizard or click on the icon in the Radiant software toolbar. The wizard leads you through a series of steps that include selecting a simulation project name, location, specifying the simulator type, selecting the process stage to use, and selecting the source files. To learn more about the flow, view “Simulation Wizard Flow” on page 71.

Source Template

The Source Template is a project view that provides templates for creating VHDL, Verilog, and constraint files. Templates increase the speed and accuracy of design entry. You can drag and drop a template directly to the source file. You can also create your own templates. For more information, view “Source Template” on page 31.

IP Catalog

IP Catalog is an easy way to use a collection of functional blocks from Lattice Semiconductor. There are two types of functional blocks available through IP Catalog: Modules and IP. IP Catalog enables you to extensively customize these blocks. They can be created as part of a specific project or as a library for multiple projects.
Modules: These basic, configurable blocks come with IP Catalog. They provide a variety of functions including I/O, arithmetic, memory, and more. Open IP Catalog to see the full list of what's available.

IP: Intellectual property (IP) are more complex, configurable blocks. They are accessible through IP Catalog, but they do not come with the tool. They must first be downloaded and installed in a separate step before they can be accessed from IP Catalog.

Overview of the IP Catalog Process  Below are the basic steps of using IP Catalog modules and IP.

1. Open IP Catalog. IP Catalog is accessed via a tab at the lower left of the Radiant software. Click the tab to view the list of available modules and IP.

2. Customize the module/IP. These modules and IP can be extensively customized for your design. The options may range from setting the width of a data bus to selecting features in a communications protocol. At a minimum you need to specify the design language to use for the output.

3. Generate the module/IP and bring its .ipx file into your project. Prior to generating the module/IP, select the option “Insert to project.” This will then automatically bring the .ipx file into your project after the generation step completes. If you do not select this option, add the .ipx file to your project after generation as you would with any other source file (such as a Verilog or VHDL file).

4. Instantiate the module/IP into the project's design. An HDL instance template is created during generation to simplify this step.

5. IP Catalog modules and IP can be further modified or updated later. After the .ipx file has been added to the Radiant software project, it is visible in the project's file list. Double-clicking the .ipx file brings up the module/IP's configuration dialog box where changes can be made and the generation process repeated.

For more information on IP Catalog, see Designing with Modules in the Radiant software online Help.

IP Packager

IP Packager allows external Intellectual Property (IP) developers -- including third party IP providers and customers -- to prepare and package IP in the Radiant software IP format.

IP packages must contain certain files, including:

- Metadata file(s) (*.xml)
WORKING WITH TOOLS AND VIEWS  :  Common Tasks

**Common Tasks**

The Radiant software gathers the many FPGA implementation tools into one central design environment. This gives you common controls for active tools, and it provides shared data between views.

**Controlling Tool Views**

Tool views are highly configurable in the Radiant software environment. You can detach a tool view to work with it as a separate window, or create tab groups to display two views side-by-side.

**Detaching and Attaching a Tool View**

Each Radiant software integrated tool view contains a Detach button in the upper-right corner that allows you to work with the tool view as a separate window.

After a tool view is detached, the Detach button changes to an Attach button, which reintegrates the view into the Radiant software main window.

You can detach as many tool views as desired. The Window menu keeps track of all open tool views and allows you to reintegrate one or all of them with the main window or detach any of them. Those that are already integrated are displayed with a check mark.

- RTL file(s) (*.v)
- Constraint template file (.ldc)
- Plugin file(s) (.py)
- Documentation files(s) (.htm, .html)
- License Agreement (*txt)

IP Packager is a standalone tool.

**To run IP Packager:**

- In Windows, go to the Windows Start menu and choose **Programs > Lattice Radiant Software > Accessories > IP Packager**.
- In Linux: from the ./<Radiant Software Install Path>/bin/lin64 directory, enter the following on a command line:
  ```
  ./ippackager
  ```

For more information on IP Packager, see *Running Radiant IP Packager* in the Radiant software online Help.
Tab Grouping

The Radiant software allows you to split one or more active tools into a separate tab group. Use the Window menu or the toolbar buttons to create the tab group and control the display.

The Split Tab Group command separates the currently active tool into a separate tab group. Having two separate tab groups enables you to work with two tool views side-by-side. This is especially useful for dragging and dropping to make constraint assignments; for example, dragging a port from Netlist View to Package View in order to assign it to a pin.

Having two separate tab groups is also useful for examining the same data element in two different views, such as the Floorplan and Physical View layouts.
Move an active tool view from one tab group to another by dragging and dropping it, or you can use the Move to Another Tab Group command.

To switch the positions of the two tab groups, click the Switch Tab Group Position command.

To merge the split tab group back into the main group, click the Merge Tab Group command.

**Using Zoom Controls**

The Radiant software includes display zoom controls in the View toolbar. There are controls for increasing or reducing the scale of the view, fitting the display contents to the window view area, and fitting a selected area or object to the window view area.

Use the mouse to quickly zoom in, out or pan graphical views, such as Floorplan View and Physical View, by doing the following:

- **Zoom In**: press and hold the left mouse button while dragging the mouse down from upper right to left to zoom in, as shown in Figure 90.
- **Zoom Out**: press and hold the left mouse button while dragging the mouse up from lower left to right to zoom out, as shown in Figure 90.
- **Zoom To**: press and hold the left mouse button while dragging the mouse down from upper left to right to zoom into the box created, as shown in Figure 91.
- **Zoom Fit**: press and hold the left mouse button while dragging the mouse up from lower right to left to reset the diagram so it fits on screen, as shown in Figure 91.
- **Pan**: Click **Pan (Enter)** and drag the mouse in any direction to move the diagram, or press and hold Ctrl and drag the mouse.
Displaying Tool Tips

When you place the cursor over a graphical element in a tool view, a tool tip appears with information on the element. The same information displayed in the tool tip will also be temporarily shown in the status bar on the lower left of the main window.
Setting Display Options

The Options dialog box, which is available from the Tools menu, enables you to specify general environment options as well as customize the display for the different tools. Tool options include selections for color, font and other graphic elements.

Color settings -- allows you to set colors for such GUI features as fonts and backgrounds for various Radiant software tools. You can also change the Theme color of the Radiant software (Dark or Light) using the Themes drop-down menu.

Tool Options

For more information about Options, refer to “Environment and Tool Options” on page 187.
Command Line Reference Guide

This help guide contains information necessary for running the core design flow development from the command line. For tools that appear in the Radiant software graphical user interface, use Tcl commands to perform commands that are described in the “Tcl Command Reference Guide” on page 150.

Command Line Program Overview

Lattice FPGA command line programs can be referred to as the FPGA flow core tools. These are tools necessary to run a complete design flow and are used for tasks such as module generation, design implementation, design verification, and design configuration. This topic provides an overview of those tools, their functions, and provides links to detailed usage information on each tool.

Each command line program provides multiple options for specifying device information, applying special functions using switches, designating desired output file names, and using command files. The programs also have particular default behavior often precludes the need for some syntax, making commands less complex. See “Command Line General Guidelines” on page 102 and “Command Line Syntax Conventions” on page 103.”
To learn more about the applications, usage, and syntax for each command line program, click on the hyperlink of the command line name in the section below.

**Note**

Many of the command line programs described in this topic are run in the background when using the tools you run in the Radiant software environment. Please also note that in some cases, command line tools described here are used for earlier FPGA architectures only, are not always recommended for command line use, or are only available from the command line.

**Design Implementation Using Command Line Tools**  The table below shows all of the command line tools used for various design functions, their graphical user interface counterparts, and provides functional descriptions of each.

**Table 1: The Radiant Software Core Design and Tool Chart**

<table>
<thead>
<tr>
<th>Design Function</th>
<th>Command Line Tool</th>
<th>Radiant Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Implementation and Auxiliary Tools</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encryption</td>
<td>Encryption</td>
<td>Encrypt Verilog/VHDL files</td>
<td>Encrypts the input HDL source file with provided encryption key.</td>
</tr>
<tr>
<td>Synthesis</td>
<td>Synpwrap</td>
<td>Synthesis Design</td>
<td>Used to manage Synopsys Synplify Pro synthesis programs.</td>
</tr>
<tr>
<td>Mapping</td>
<td>MAP</td>
<td>Map Design</td>
<td>Converts a design represented in logical terms into a network of physical components or configurable logic blocks.</td>
</tr>
<tr>
<td>Placement and Routing</td>
<td>PAR</td>
<td>Place &amp; Route Design</td>
<td>Assigns physical locations to mapped components and creates physical connections to join components in an electrical network.</td>
</tr>
<tr>
<td>Static Timing Analysis</td>
<td>Timing</td>
<td>Post-Synthesis Timing Report, MAP Timing Report, Place &amp; Route Timing Report</td>
<td>Generates reports that can be used for static timing analysis.</td>
</tr>
<tr>
<td>Back Annotation</td>
<td>Backanno</td>
<td>Tool does not exist in the Radiant software interface as process but employed in file export.</td>
<td>Distributes the physical design information back to the logical design to generate a timing simulation file.</td>
</tr>
</tbody>
</table>
Table 1: The Radiant Software Core Design and Tool Chart

<table>
<thead>
<tr>
<th>Design Function</th>
<th>Command Line Tool</th>
<th>Radiant Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitstream Generation</td>
<td>BITGEN</td>
<td>Bitstream</td>
<td>Converts a fully routed physical design into configuration bitstream data.</td>
</tr>
<tr>
<td>Device Programming</td>
<td>PGRCMD</td>
<td>Device Programming</td>
<td>Downloads data files to an FPGA device.</td>
</tr>
<tr>
<td>IP Packager</td>
<td>IPPKG</td>
<td>IP Packager</td>
<td>IP author select files from disks and pack them into one IPK file.</td>
</tr>
</tbody>
</table>

**Command Line Basics**

This section provides basic instructions for running any of the core design flow development and tools from the command line.

Topics include:

- “Command Line Data Flow” on page 101
- “Command Line General Guidelines” on page 102
- “Command Line Syntax Conventions” on page 103
- “Invoking Core Tool Command Line Programs” on page 105
- “Setting Up the Environment to Run Command Line” on page 104
- “Invoking Core Tool Command Line Tool Help” on page 105

**Command Line Data Flow**

The following chart illustrates the FPGA command line tool data flow through a typical design cycle.

**Command Line Tool Data Flow**

See Also  ►“Command Line Reference Guide” on page 99

► “Command Line General Guidelines” on page 102
Command Line General Guidelines

You can run the FPGA family Radiant software design tools from the command line. The following are general guidelines that apply.

- Files are position-dependent. Generally, they follow the convention [options] <infile> <outfile> (although order of <outfile> and <infile> are sometimes reversed). Use the -h command line option to check the exact syntax; for example, par -h.

- For any Radiant software FPGA command line program, you can invoke help on available options with the -h or -help command. See “Invoking Core Tool Command Line Programs” on page 105 for more information.

- Command line options are entered on the command line in any order, preceded by a hyphen (-), and separated by spaces.

- Most command line options are case-sensitive and must be entered in lowercase letters. When an option requires an additional parameter, the option and the parameter must be separated by spaces or tabs (i.e., -l 5 is correct, -l5 is not).

- Options can appear anywhere on the command line. Arguments that are bound to a particular option must appear after the option (i.e., -f <command_file> is legal; <command_file> -f is not).

- For options that may be specified multiple times, in most cases the option letter must precede each parameter. For example, -b romeo juliet is not acceptable, while -b romeo -b juliet is acceptable.

- If you enter the FPGA family Radiant software application name on the command line with no arguments and the application requires one or more arguments (par, for example), you get a brief usage message consisting of the command line format string.
For any Radiant software FPGA command line program, you can store program commands in a command file. Execute an entire batch of arguments by entering the program name, followed by the \texttt{-f} option, and the command file name. This is useful if you frequently execute the same arguments each time you execute a program or to avoid typing lengthy command line arguments. See “Using Command Files” on page 145.

See Also

- “Invoking Core Tool Command Line Tool Help” on page 105
- “Command Line Syntax Conventions” on page 103
- “Using Command Files” on page 145
- “Command Line Data Flow” on page 101

## Command Line Syntax Conventions

The following conventions are used when commands are described:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Encloses a logical grouping for a choice between sub-formats.</td>
</tr>
<tr>
<td>[]</td>
<td>Encloses items that are optional. (Do not type the brackets.) Note that &lt;infile.udb&gt; indicates that the .udb extension is optional but that the extension must be UDB.</td>
</tr>
<tr>
<td>{}</td>
<td>Encloses items that may be repeated zero or more times.</td>
</tr>
<tr>
<td></td>
<td>Logical OR function. You must choose one or a number of options. For example, if the command syntax says \texttt{pan up</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Encloses a variable name or number for which you must substitute information.</td>
</tr>
<tr>
<td>, (comma)</td>
<td>Indicates a range for an integer variable.</td>
</tr>
<tr>
<td>- (dash)</td>
<td>Indicates the start of an option name.</td>
</tr>
<tr>
<td>:</td>
<td>The bind operator. Binds a variable name to a range.</td>
</tr>
<tr>
<td><strong>bold</strong> text</td>
<td>Indicates text to be taken literally. You type this text exactly as shown (for example, “Type \texttt{autoroute -all -i 5} in the command area.”) Bold text is also used to indicate the name of an EPIC command, a Linux command, or a DOS command (for example, “The \texttt{playback} command is used to execute the macro you created.”).</td>
</tr>
</tbody>
</table>
Setting Up the Environment to Run Command Line

For Windows  The environments for both the Radiant Tcl Console window or Radiant Standalone Tcl Console window (pnmainc.exe) are already set. You can start entering Tcl tool command or core tool commands in the console and the software will perform them.

When running the Radiant software from the Windows command line (via cmd.exe), you will need to add the following values to the following environment variables:

- **PATH includes**, for 64-bit
  
  `<Install_directory>in\nt64; <Install_directory>\ispfpga\bin\nt64`
  
  **Example** `<Install_directory>`:
  
  `c:\lscc\radiant\1.0\bin\nt64; c:\lscc\radiant\1.0\ispfpga\bin\nt64`

- **FOUNDARY includes**
  
  `set FOUNDARY= <Install_directory>\ispfpga`

For Linux  On Linux, the Radiant software provides a similar standalone Tcl Console window (radiantc) that sets the environment. The user can enter Tcl commands and core tool commands in it.

If you do not use the Tcl Console window, you need to run "bash" to switch to BASH" first, then run the following command.

- **For BASH** (64-bit):
After setting up for either Windows or PC, you can run the Radiant software executable files directly. For example, you can invoke the Place and Route program by:

```bash
par test_map.udb test_par.udb
```

See Also
- "Invoking Core Tool Command Line Programs" on page 105
- "Invoking Core Tool Command Line Tool Help" on page 105

**Invoking Core Tool Command Line Programs**

This topic provides general guidance for running the the Radiant software FPGA flow core tools. Refer to "Command Line Program Overview" on page 99 to see what these tools include and for further information.

For any the Radiant software FPGA command line programs, you begin by entering the name of the command line program followed by valid options for the program separated by spaces. Options include switches (-f, -p, -o, etc.), values for those switches, and file names, which are either input or output files. You start command line programs by entering a command in the Linux™ or DOS™ command line. You can also run command line scripts or command files.

See Table 2 on page 103 for details and links to specific information on usage and syntax. You will find all of the usage information on the command line in the Running FPGA Tools from the Command Line > Command Line Tool Usage book topics.

See Also
- "Command Line Syntax Conventions" on page 103
- "Invoking Core Tool Command Line Tool Help" on page 105
- "Setting Up the Environment to Run Command Line" on page 104
- "Using Command Files" on page 145

**Invoking Core Tool Command Line Tool Help**

To get a brief usage message plus a verbose message that explains each of the options and arguments, enter the FPGA family Radiant software application name on the command line followed by -help or -h. For example, enter `bitgen -h` for option descriptions for the `bitgen` program.
To redirect this message to a file (to read later or to print out), enter this command:

```
command_name -help | -h > filename
```

The usage message is redirected to the filename that you specify.

For those FPGA family Radiant software applications that have architecture-specific command lines (e.g., iCE UltraPlus), you must enter the application name, -help (or -h), and the architecture to get the verbose usage message specific to that architecture. If you fail to specify the architecture, you get a message similar to the following:

Use `<appname> -help <architecture>` to get detailed usage for a particular architecture.

See Also

- “Command Line Data Flow” on page 101
- “Command Line General Guidelines” on page 102
- “Command Line Syntax Conventions” on page 103
- “Setting Up the Environment to Run Command Line” on page 104
- “Using Command Files” on page 145

Command Line Tool Usage

This section contains usage information of all of the command line tools and valid syntax descriptions for each.

Topics include:

- “Running cmpl_libs.tcl from the Command Line” on page 107
- “Running HDL Encryption from the Command Line” on page 109
- “Running SYNTHESIS from the Command Line” on page 116
- “Running Postsyn from the Command Line” on page 122
- “Running MAP from the Command Line” on page 123
- “Running PAR from the Command Line” on page 125
- “Running Timing from the Command Line” on page 131
- “Running Backannotation from the Command Line” on page 133
- “Running Bit Generation from the Command Line” on page 136
- “Running Various Utilities from the Command Line” on page 141
- “Using Command Files” on page 145
- “Using Command Line Shell Scripts” on page 147
Running cmpl_libs.tcl from the Command Line

The cmpl_libs.tcl command allows you to perform simulation library compilation from the command line.

The following information is for running cmpl_libs.tcl from the command line using the tclsh application. The supported TCL version is 8.5 or higher.

If you don't have TCL installed, or you have an older version, perform the following:

1. Add `<Radiant_install_path>/tcltk/windows/BIN` to the front of your PATH, and
2. For Linux users only, add `<Radiant_install_path>/tcltk/linux/bin` to the front of your LD_LIBRARY_PATH

**Note**
The default version of TCL on Linux could be older and may cause the script to fail. Ensure that you have TCL version 8.5 or higher.

To check TCL version, type:

```
tclsh
% info tclversion
% exit
```

For script usage, type:

```
tclsh cmpl_libs.tcl [-h|-help]
```

**Notes**

1. If Modelsim/Questa is already in your PATH and preceding any Aldec tools, you can use:
   
   `-sim_path .` for simplification; `.` will be added to the front of your PATH.
2. Ensure the FOUNDRY environment variable is set. If the FOUNDRY environment variable is missing, then you need to set it before running the script. For details, refer to “Setting Up the Environment to Run Command Line” on page 96.
3. To execute this script error free, Questasim 10.4e or a later 10.4 version, or Questasim 10.5b or a later version should be used for compilation.

Check log files under `<target_path>` (default = .) for any errors, as follows:

1. For Linux, type:
   
   `grep -i error *.log`

2. For Windows, type:
   
   `find /i "error" *.log`
Subjects included in this topic:

- Running compl_lib.tcl
- Command Line Syntax
- cmpl_libs.tcl Options
- Examples

Running compl_lib.tcl  cmpl_libs.tcl allows you to compile simulation libraries from the command line.

Command Line Syntax  tclsh <Radiant_install_path>/cae_library/simulation/scripts/cmpl_libs.tcl -sim_path <sim_path> [-sim_vendor {mentor<default>}] [-device {ice40up|all<default>}] [-target_path <target_path>]

cmpl_libs.tcl Options  The table below contains all valid options for cmpl_libs.tcl

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-sim_path &lt;sim_path&gt;</td>
<td>The -sim_path argument specifies the path to the simulation tool executable (binary) folder. This option is mandatory. Currently only Modelsim and Questa simulators are supported. NOTE: If &lt;sim_path&gt; has spaces, then it must be surrounded by &quot; &quot;. Do not use { }.</td>
</tr>
<tr>
<td>[-sim_vendor {mentor&lt;default&gt;}]</td>
<td>The -sim_vendor argument is optional, and intended for future use. It currently supports only Mentor Graphics simulators (Modelsim / Questa).</td>
</tr>
<tr>
<td>[-device {ice40up</td>
<td>all&lt;default&gt;}]</td>
</tr>
<tr>
<td>[-target_path &lt;target_path&gt;]</td>
<td>The -target_path argument specifies the target path, where you want the compiled libraries and modelsim.ini file to be located. This argument is optional, and the default target path is the current folder. NOTES: (1) This argument is recommended if the current folder is the Radiant software's startup (binary) folder, or if the current folder is write-protected. (2) If &lt;target_path&gt; has spaces, then it must be surrounded by &quot; &quot;. Do not use { }.</td>
</tr>
</tbody>
</table>

Examples  This section illustrates and describes a few examples of Simulation Libraries Compilation Tcl command.

Example 1  The following command will compile all the Lattice FPGA libraries for Verilog simulation, and place them under the folder specified by -target_path. The path to Modelsim is specified by -sim_path.
Running HDL Encryption from the Command Line

Radiant software allows you to encrypt the individual HDL source files.

The tool supports encryption of Verilog HDL and VHDL files. Per command’s execution, single source file is encrypted.

The HDL file can be partially or fully encrypted depending on pragmas’ placements within the HDL file. To learn more about pragmas’ placements, see “Defining Pragmas” on page 111.

Running HDL Encryption   Before running the utility, you need to annotate the HDL file with the appropriate pragmas. Additionally, you may need to create a key file containing an encryption key. To view the key file’s proper formatting, see “Key File” on page 115.

Command Line Syntax  encrypt_hdl [-k <keyfile>] [-l language] [-o <output_file>] <input_HDL_file>

See Also  ➤“Command Line Program Overview” on page 99
Encryption Option  The table below contains descriptions of all valid options for HDL encryption.

### Table 4: Encryption Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h(elp)</td>
<td>Print command help message.</td>
</tr>
<tr>
<td>-k &lt;keyfile&gt;</td>
<td>A key repository file. Depending on the location of the key, this option is required or optional.</td>
</tr>
<tr>
<td></td>
<td>▶ If the HDL source file contains no pragma, the key file is required. The tool encrypts the entire HDL file using all key sets declared in the key file.</td>
</tr>
<tr>
<td></td>
<td>▶ If the HDL source file contains only <code>begin</code> and <code>end</code> pragmas and no key pragmas, the key file is required. The tool encrypts the section between <code>begin</code> and <code>end</code> using all key sets declared in the key file.</td>
</tr>
<tr>
<td></td>
<td>▶ If the HDL source file contains the proper key pragma, <code>key_keyowner</code>, <code>key_keyname</code>, but the key file is missing the provided <code>key_public_key</code>, the tool fetches the first public key string matching the <code>key_keyowner</code> and <code>key_keyname</code> requirement in the key file.</td>
</tr>
<tr>
<td></td>
<td>If the HDL source file contains the proper definition of key, this option is not required.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>If the same key name is defined in both, HDL source file and key.txt file, the key defined in HDL source file has a precedence.</td>
</tr>
<tr>
<td>-l &lt;language&gt;</td>
<td>Directive language, vhdl or verilog (default).</td>
</tr>
<tr>
<td>-o &lt;output_file&gt;</td>
<td>An encrypted HDL file. This is an optional field. If not defined during the encryption, the tool generates a new output file <code>&lt;input_file_name&gt;_enc.v</code>.</td>
</tr>
</tbody>
</table>

#### Examples

This section illustrates and describes a few examples of HDL encryption using Tcl command.

**Example 1:** This example shows a successful encryption of HDL file with default options. It is assumed that key is properly defined in HDL file. Since output file name was not specified, the tool generates an output file `<file_name>_enc.v` in the same directory as the location of the input file.

```
> encrypt_hdl -k source/impl_1/keys.txt -o top.v top.v
Options:
Key repository file: source/impl_1/keys.txt
Directive language: <not specified>, use verilog as default
Output file: top.v
Processed 2 envelopes.
```

**Example 2:** This example shows a successful encryption of HDL file by generating a new output file.

```
> encrypt_hdl -k source/impl_1/keys.txt -o remote_files/top_vl_enc.v remote_files/sources/top_vl_part.
Options:
Key repository file: source/impl_1/keys.txt
Directive language: <not specified>, use verilog as default
Output file: remote_files/top_vl_enc.v
Processed 2 envelopes.
```
Example3:  This example shows unsuccessful HDL encryption due to a missing key file. To correct this issue, the user must either define the appropriate key file key.txt or annotated the HDL file with appropriate pragmas. To correct the issue, define the key either in key.txt file or directly in HDL source file.

```plaintext
> encrypt_hdl -o remote_files/sources/top_v1_part_en.v remote_files/sources/top_v1_part.v
Options:
  Key repository file: <not specified>
  Directive language: <not specified>, use verilog as default
  Output file: remote_files/sources/top_v1_part_en.v
ERROR - remote_files/sources/top_v1_part.v at line 68: missing key.
```

**NOTE**

A key is always required in the encryption tool while key file is optional. If the complete key: `key_keyowner, key_keyname, key_method, and key_public_key`, is defined within HDL source file, key file is not required.

For specific steps and information on how to encrypt HDL files in the Radiant software, refer to the following section in the Radiant software online help: 
**User Guides > Securing the Design.**

### Defining Pragmas

Pragma are used to specify the portion of the HDL source file that must be encrypted.Pragma's definition is compliant with IEEE 1735-2014 V1 standard.

Pragma syntax in Verilog HDL file:

```plaintext`
'pragma protect <pragma’s option>
```

Pragma synax in VHDL file:

```plaintext`
'protect <pragma’s option>
```

#### Table 5: List of available Pragma Options

<table>
<thead>
<tr>
<th>Name</th>
<th>Available Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>1 (default)</td>
<td>Specifies the current Radiant software encryption version.</td>
</tr>
<tr>
<td>author</td>
<td>string</td>
<td>Specifies the file creator.</td>
</tr>
<tr>
<td>author_info</td>
<td>string</td>
<td>Additional information you would like to include in file.</td>
</tr>
<tr>
<td>encoding</td>
<td>base64</td>
<td>The output format of processed data.</td>
</tr>
<tr>
<td>begin</td>
<td></td>
<td>The start point for data obfuscation.</td>
</tr>
<tr>
<td>end</td>
<td></td>
<td>The end point for data obfuscation.</td>
</tr>
<tr>
<td>key_keyowner</td>
<td>string</td>
<td>The key creator.</td>
</tr>
<tr>
<td>key_keyname</td>
<td>string</td>
<td>The RSA key name to specify the private key.</td>
</tr>
<tr>
<td>key_method</td>
<td>rsa</td>
<td>The cryptographic algorithm used for key obfuscation.</td>
</tr>
</tbody>
</table>
To encrypt HDL source file, encryption version, encoding type, and key specific pragmas must be defined in the HDL source file by HDL designer; only the content within the pragmas is encrypted.

**NOTE**

Multiple key sets can be declared in a single key file.

### Example of Verilog source file marked with Pragmas:

```
// 3 bit counter with asynchronous reset
module count (c,clk, rst);

input clk, rst;
output [2:0] c;
reg [2:0] c;

    //pragma protect version = 1
    //pragma protect author = "<Your Name>">
    //pragma protect author_info = "<Your info>">
    //pragma protect key_keyowner = "Lattice Semiconductor"
    //pragma protect key_keyname = "LATTICE_RADIANT_2"
    //pragma protect key_method = "rsa"
    //pragma protect key_public_key

    //put a blank line above
    //pragma protect data_method = "aes256-cbc"

    //pragma protect begin
    always @ (posedge clk or posedge rst)
    begin
      if (rst)
        c = 3'b000;
      else
        c = c + 1;
    end
    //pragma protect end

endmodule
```

The encrypted file may contain multiple encrypted key sets.

| Table 5: List of available Pragma Options |
|-----------------------------------------|--------------------------------------|
| Name                  | Available Values | Description |
| key_public_key        | aes128-cbc       | The RSA public key file name. |
|                       | aes256-cbc (default) | The AES encryption data method. |

To encrypt HDL source file, encryption version, encoding type, and key specific pragmas must be defined in the HDL source file by HDL designer; only the content within the pragmas is encrypted.
Example of encrypted Verilog file:

```verilog
// 3 bit counter with asynchronous reset
module count (p_clk reset);
input clk, reset;
output [2:0] z;
reg [2:0] c;

// put a blank line above

'pragma protect begin protected
'pragma protect version 1
'pragma protect author "Your Name"
'pragma protect author info "Your info"
'pragma protect encrypt agent "Radiant encrypt_bdl"
'pragma protect encrypt agent info "Radiant encrypt_bdl Version 1.0"

'pragma protect encoding (encrypt_type = "base64", line_length = 64, byte_length = 256)
'pragma protect key_keyname "Lattice Semiconductor"
'pragma protect key_keyname "Lattice Semiconductor"
'pragma protect key_length 64
'pragma protect key_method "rsa"
'pragma protect key_block

// encrypted data
//...
//...
//pragma protect end protected

endmodule
```
Example of VHDL source file marked by Pragmas:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_2304.all;
use ieee.std_logic_arith.all;

entity top_test is
    port(
        data : out std_logic_vector(7 downto 0);
        reset : in std_logic;
        clk : in std_logic
    );
end top_test;
architecture top_test_arch of top_test is
signal count : std_logic_vector(7 downto 0);
begin
    process (version-1)
    begin
        if (version = '1') then
            count <= '0';
        else
            if (clk'event and clk = '1') then
                count <= count + 1;
            end if;
        end if;
    end process;
end top_test_arch;
```
Example of encrypted VHDL file:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity top_test is

    port(
        ocnt : out std_logic_vector(7 downto 0);
        reset : in std_logic;
        clk : in std_logic
    );
end top_test;

architecture top_test_arch of top_test is

signal ocnt : std_logic_vector(7 downto 0);

begin

  process
  begin
    wait;
    ocnt <= "00000000";
  end process;

end top_test_arch;
```

See Also
- “Running HDL Encryption from the Command Line” on page 109
- “Key File” on page 115

**Key File**

The key repository file defines the cryptographic public key used for RSA encryption. In Radiant software, the key file contains Lattice public key. Additionally, it may contain some of the common EDA vendors public keys.
The Lattice public key file key.txt is located at `<Radiant_installed_directory>/ispfpga/data/` folder. Aside of Lattice public key, the current version contains the public key for Synopsys, Aldec, and Cadence.

**NOTE**

If using Synplify Pro synthesis tool, both, the Lattice Public Key and the Synplify Pro Public Key must be defined in the key file. The Synplify Pro Public Key is used during the synthesis step to decrypt an encrypted design. The Lattice Public Key is used during the post-synthesis flow to decrypt an encrypted design.

A key file must contain properly declared pragmas such as `key_keyowner`, `key_keyname`, `key_method`, and `key_public_key` for each of the specified keys. The key value follows the `key_public_key` pragma.

The key file typically also contains the `data_method` pragma. It defines the algorithm used in data block encryption of HDL source file.

**Example of a Key File:**

```plaintext
// Use Verilog pragmas syntax in this file
pragma protect version=1
pragma protect author="<Your Name>",<your info>
pragma protect key_keyowner="Lattice Semiconductor"
pragma protect key_keyname="LSCC_RADIANT_2"
pragma protect key_method="rsa"
pragma protect key_public_key M12812NH2gwpk19s<0L0MFAQQ/7h/QiwK6QMI50qKqOAQmKCTHv3Nhsetk6vJae7GbQv
1N6AM5usom\O9p\L1F71Ae13q9o7Y201Xkha1a9GTea5Pf3bxcexChu<5r0n5h
3X2S)f0BQ12ozu2b7E20XhAYKJvWX87608eKLH2DQ9zb0vh1D0z2qE26n2
IaE2f3pypT4x/hn0e6XiQo567gup000h000c1222ly5XQx42qV290y10c
0c/sofx5f9f5h096Q08g--xz1z0zcxzg99996x12424542<79yc57
sd1h0d2q02lljelz2z942mgL2J4Dv-01e2Ph3d011/N30y432c6h9fn04zKzKz7
Twf3Qck8

// Put a blank line above
// Add additional public keys below this line
pragma protect data_method="aes/256-128"

// End of File
```

**See Also**

- “Running HDL Encryption from the Command Line” on page 109
- “Defining Pragmas” on page 111

**Running SYNTHESIS from the Command Line**

The Lattice synthesis tool SYNTHESIS allows you to synthesize Verilog and VHDL HDL source files into netlists for design entry into the Radiant software environment. Based on your strategy settings you specify in the Radiant software, a synthesis project (.synproj) file is created and then used by SYNTHESIS using the `-f` option. The Radiant software translates strategy options into command line options described in this topic.

Verilog source files are passed to the program using the `-ver` option and VHDL source files are passed using the `-vhd` option. For mixed language
designs the language type is automatically determined by SYNTHESIS based on the top module of the design. For IP design, you must also specify IP location (-ip_dir), IP core name (-corename), and encrypted RTL file name (-ertl_file).

Subjects included in this topic:
- Running SYNTHESIS
- Command Line Syntax
- SYNTHESIS Options
- Examples

Running SYNTHESIS   SYNTHESIS will convert your input netlist (.v) file into a structural verilog file that is used for the remaining mapping process.
- To run SYNTHESIS, type synthesis on the command line with valid options. A sample of a typical SYNTHESIS command would be as follows:

There are many command line options that give you control over the way SYNTHESIS processes the output file. Please refer to the rest of the subjects in this topic for more details. See examples.

**SYNTHESIS Options**  The table below contains descriptions of all valid options for SYNTHESIS.

### Table 6: SYNTHESIS Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a &lt;arch&gt;</code></td>
<td>Sets the FPGA architecture. This synthesis option must be specified and if the value is set to any unsupported FPGA device architecture the command will fail.</td>
</tr>
<tr>
<td><code>-p &lt;device&gt;</code></td>
<td>Specifies the device type for the architecture (optional).</td>
</tr>
<tr>
<td><code>-f &lt;proj_file_name&gt;</code></td>
<td>Specifies the synthesis project file name (.synproj). The project file can be edited by the user to contain all desired command line options.</td>
</tr>
<tr>
<td><code>-t &lt;package_name&gt;</code></td>
<td>Specifies the package type of the device.</td>
</tr>
<tr>
<td><code>-path &lt;searchpath&gt;</code></td>
<td>Add searchpath for Verilog &quot;include&quot; files (optional).</td>
</tr>
<tr>
<td><code>-top &lt;top_module_name&gt;</code></td>
<td>Name of top module (optional, but better to have to avoid ambiguity).</td>
</tr>
<tr>
<td><code>-lib &lt;lib_name&gt;</code></td>
<td>Name of VHDL library (optional).</td>
</tr>
<tr>
<td><code>-vhd &lt;vhdl_file.vhd/vhdl&gt;</code></td>
<td>Names of VHDL design files (must have, if language is VHDL or mixed language).</td>
</tr>
<tr>
<td><code>-ver &lt;verilog_file.v&gt;</code></td>
<td>Names of Verilog design files (must have, if language is Veril, or mixed language).</td>
</tr>
<tr>
<td><code>-hdl_param &lt;name, value&gt;</code></td>
<td>Allows you to override HDL parameter pairs in the design file.</td>
</tr>
<tr>
<td>`-optimization_goal &lt;balanced (default)</td>
<td>area</td>
</tr>
<tr>
<td></td>
<td>▶ balanced balances the levels of logic.</td>
</tr>
<tr>
<td></td>
<td>▶ area optimizes the design for area by reducing the total amount of logic used for design implementation.</td>
</tr>
<tr>
<td></td>
<td>▶ timing optimizes the design for timing.</td>
</tr>
<tr>
<td></td>
<td>The default setting depends on the device type. Smaller devices, such as ice40tp default to balanced.</td>
</tr>
<tr>
<td>`-force_gsr &lt;auto</td>
<td>yes</td>
</tr>
</tbody>
</table>
Sets the type of random access memory globally to distributed, embedded block RAM, or registers. The default is auto which attempts to determine the best implementation, that is, synthesis tool will map to technology RAM resources (EBR/Distributed) based on the resource availability.

This option will apply a syn_ramstyle attribute globally in the source to a module or to a RAM instance. To turn off RAM inference, set its value to registers.

- **registers** causes an inferred RAM to be mapped to registers (flip-flops and logic) rather than the technology-specific RAM resources.
- **distributed** causes the RAM to be implemented using the distributed RAM or PFU resources.
- **block_ram(EBR)** causes the RAM to be implemented using the dedicated RAM resources. If your RAM resources are limited, for whatever reason, you can map additional RAMs to registers instead of the dedicated or distributed RAM resources using this attribute.
- **no_rw_check** (Certain technologies only). You cannot specify this value alone. Without no_rw_check, the synthesis tool inserts bypass logic around the RAM to prevent the mismatch. If you know your design does not read and write to the same address simultaneously, use no_rw_check to eliminate bypass logic. Use this value only when you cannot simultaneously read and write to the same RAM location and you want to minimize overhead logic.
COMMAND LINE REFERENCE GUIDE  :  Command Line Tool Usage

Table 6: SYNTHESES Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -romstyle <auto (default) | Allows you to globally implement ROM architectures using dedicated, distributed ROM, or a combination of the two (auto). This applies the syn_romstyle attribute globally to the design by adding the attribute to the module or entity. You can also specify this attribute on a single module or ROM instance. Specifying a syn_romstyle attribute globally or on a module or ROM instance with a value of:  
  - auto allows the synthesis tool to choose the best implementation to meet the design requirements for performance, size, etc.  
  - EBR causes the ROM to be mapped to dedicated EBR block resources. ROM address or data should be registered to map it to an EBR block. If your ROM resources are limited, for whatever reason, you can map additional ROM to registers instead of the dedicated or distributed RAM resources using this attribute. Infer ROM architectures using a CASE statement in your code. For the synthesis tool to implement a ROM, at least half of the available addresses in the CASE statement must be assigned a value. For example, consider a ROM with six address bits (64 unique addresses). The case statement for this ROM must specify values for at least 32 of the available addresses. |

- -output_hdl <filename.v> Specifies the name of the output Verilog netlist file.  
- -sdc <sdc_file.ldc> Specifies a Lattice design constraint (.ldc) file input.  
- -loop_limit <max_loop_iter_cnt (default 1950)> Specifies the iteration limits for “for” and “while” loops in the user RTL for loops that have the loop index as a variable and not a constant. The higher the loop_limit, the longer the run time. Also, for some designs, a higher loop limit may cause stack overflow during some of the optimizations during compile/synthesis. The default value is 1950. Setting a higher value may cause stack overflow during some of the optimizations during synthesis.  
- -logfile <synthesis_logfile> Specifies the name of the synthesis log file in ASCII format. If you do not specify a name, SYNTHESES will output a file named synthesis.log by default.  
- -frequency <target_frequency (default 200.0MHz (ICE40))> Specifies the target frequency setting. Default frequency value is 200.0 MHz.  
- -max_fanout <value> Specifies maximum global fanout limit to the entire design at the top level. Default value is 1000 fanouts.  
- -bram_utilization <value> Specifies block RAM utilization target setting in percent of total vacant sites. Default is 100 percent. |
**Table 6: SYNTHESIS Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`-fsm_encoding_style &lt;auto</td>
<td>one-hot</td>
</tr>
<tr>
<td>`-use_carry_chain &lt;0</td>
<td>1&gt;`</td>
</tr>
<tr>
<td><code>-carry_chain_length &lt;chain_length&gt;</code></td>
<td>Specifies the maximum length of the carry chain.</td>
</tr>
<tr>
<td>`-use_io_insertion &lt;0</td>
<td>1&gt;`</td>
</tr>
</tbody>
</table>
| `-use_io_reg <0|1|auto(default)>` | Packs registers into I/O pad cells based on timing requirements for the target Lattice families. The value 1 enables and 0 disables (default) register packing. This applies it globally forcing the synthesis tool to pack all input, output, and I/O registers into I/O pad cells. NOTE: You can place the syn_useioff attribute on an individual register or port. When applied to a register, the synthesis tool packs the register into the pad cell, and when applied to a port, packs all registers attached to the port into the pad cell. The syn_useioff attribute can be set on a:  
  ▶ top-level port  
  ▶ register driving the top-level port  
  ▶ lower-level port, only if the register is specified as part of the port declaration |
| `-resource_sharing <0|1>` | Specifies the resource sharing option. The 1 or true setting is the default. |
| `-propagate_constants <0|1>` | Prevents sequential optimization such as constant propagation, inverter push-through, and FSM extraction. The 1 or true setting is the default. |
| `-remove_duplicate_regs <0|1>` | Specifies the removal of duplicate registers. The 1 or true setting is the default. |
| `-twr_paths <timing_path_cnt>` | Specifies the number of critical paths. |
| `-dt` | Disables the hardware evaluation capability. |
| `-udb <udb_file.udb>` | Sets option to dump intermediate files. If you run the tool with this option, it will dump about 20 intermediate encrypted Verilog files. If you supply Lattice with these files, they can be decrypted and analyzed for problems. This option is good for analyzing simulation issues. |
Examples  Following are a few examples of SYNTHESIS command lines and a description of what each does.

```
synthesis -a "ice40tp" -p itpa08 -t SG48 -sp "6" -mux_style Auto
-use_io_insertion 1
-sdc "C:/my_radiant_tutorial/impl1/impl1.ldc"
-path "C:/lscc/radiant/1.0/ispfpga/ice40tp/data" "C:/my_radiant_tutorial/impl1" "C:/my_radiant_tutorial"
-ver "C:/my_radiant_tutorial/impl1/source/LED_control.v" "C:/my_radiant_tutorial/impl1/source/spi_gpio.v" "C:/my_radiant_tutorial/impl1/source/spi_gui_led_top.v"
-path "C:/my_radiant_tutorial"
-top spi_gui_led_top
-output_hdl "LEDtest_impl1.vm"
```

See Also  “Command Line Program Overview” on page 99

### Running Postsyn from the Command Line

The Postsyn process converts synthesized VM and integrates IPs into a completed design in UDB format for the remaining mapping process.


---

**Table 6: SYNTHESIS Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-fix_gated_clocks</code></td>
<td>Allows you to enable/disable gated clock optimization. By default, the option is enabled.</td>
</tr>
<tr>
<td><code>-vh2008</code></td>
<td>Enables VHDL 2008 support.</td>
</tr>
</tbody>
</table>

---
Running MAP from the Command Line

The Map Design process in the Radiant software environment can also be run through the command line using the map program. The map program takes an input database (.udb) file and converts this design represented as a network of device-independent components (e.g., gates and flip-flops) into a network of device-specific components (e.g., PFUs, PFFs, and EBRs) or configurable logic blocks in the form of a Unified Database (.udb) file.

Subjects included in this topic:

- Running MAP
- Command Line Syntax
- MAP Options
- Examples

Running MAP MAP uses the database (.udb) file that was the output of the Synthesis process and outputs a mapped Unified Database (.udb) file with constraints embedded.
To run MAP, type `map` on the command line with, at minimum, the required options to describe your target technology (i.e., architecture, device, package, and performance grade), the input .udb along with the input .ldc file. The output .udb file specified by the `-o` option. That additional physical constraint file (*.pdc) can be applied optionally. A sample of a typical MAP command would be as follows:

```
map counter_impl1_syn.udb impl1.pdc -o counter_impl1.udb
```

**Note**

The `-a` (architecture) option is not necessary when you supply the part number with the `-p` option. There is also no need to specify the constraint file here, but if you do, it must be specified after the input .udb file name. The constraint file automatically takes the name “output” in this case, which is the name given to the output .udb file. If the output file was not specified with the `-o` option as shown in the above case, `map` would place a file named input.udb into the current working directory, taking the name of the input file. If you specify the input.ldc file and it is not there, `map` will error out.

There are many command line options that give you control over the way MAP processes the output file. Please refer to the rest of the subjects in this topic for more details.

**Command Line Syntax**

```
map [ -h <arch> ] <infile[.udb]> [ <options> ]
```

**MAP Options**

The table below contains descriptions of all valid options for MAP.

**Table 8: MAP Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h &lt;arch&gt;</code></td>
<td>Displays all of the available MAP command options for mapping to the specified architecture.</td>
</tr>
<tr>
<td><code>&lt;infile[.udb]&gt;</code></td>
<td>Specifies the output design file name in .udb format. The .udb extension is optional.</td>
</tr>
<tr>
<td><code>-inferGSR</code></td>
<td>GSR inferencing if applicable.</td>
</tr>
<tr>
<td><code>-o &lt;name[.udb]&gt;</code></td>
<td>Optional output design file .udb.</td>
</tr>
<tr>
<td><code>-mp &lt;name[.mrp]&gt;</code></td>
<td>Optional report file (.mrp).</td>
</tr>
<tr>
<td><code>-xref_sig</code></td>
<td>Report signal cross reference for renamed signals.</td>
</tr>
<tr>
<td><code>-xref_sym</code></td>
<td>Report symbol cross reference for renamed symbols.</td>
</tr>
<tr>
<td><code>-u</code></td>
<td>Unclip unused instances.</td>
</tr>
</tbody>
</table>

**Examples**

Following are some examples of MAP command lines and a description of what each does.

**Example 1**

The following command maps an input database file named mapped.udb and outputs a mapped Unified Database file named mapped.udb.
map counter_impl1_syn.udb impl1.pdc -o counter_impl1.udb

See Also  ► “Command Line Data Flow” on page 101
► “Command Line Program Overview” on page 99

Running PAR from the Command Line

The Place & Route Design process in the Radiant software environment can also be run through the command line using the par program. The par program takes an input mapped Unified Database (.udb) file and further places and routes the design, assigning locations of physical components on the device and adding the inter-connectivity, outputting a placed and routed .udb file.

The Implementation Engine multi-tasking option available in Linux is explained in detail here because the option is not available for PCs.

Subjects included in this topic:
► Running PAR
► Command Line Syntax
► General Options
► Placement Options
► Routing Options
► PAR Explorer (-exp) Options
► Examples
► PAR Multi-Tasking Options

Running PAR  PAR uses your mapped Unified Database (.udb) file that were the outputs of the Map Design process or the map program. With these inputs, par outputs a new placed-and-routed .udb file, a PAR report (.par) file, and a PAD (specification (.pad) file that contains I/O placement information.

► To run PAR, type par on the command line with at minimum, the name of the input .udb file and the desired name of the output .udb file. “Design constraints from previous stages are automatically embedded in the input .udb file, however the par program can accept additional constraints with either a .pdc or .sdc file” A sample of a basic PAR command would be as follows:

par input.udb output.udb

There are many command line options that give you control over PAR. Please refer to the rest of the subjects in this topic for more details.

(infile) (outfile) [pdcfile]

**Note**

All filenames without special switches must be in the order <infile> <outfile> <pdcfile>. Options may exist in any order.

### General Options

**Table 9: General PAR Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f</td>
<td>Read par command line arguments and switches from file.</td>
</tr>
<tr>
<td>-w</td>
<td>Overwrite. Allows overwrite of an existing file (including input file).</td>
</tr>
<tr>
<td>-n</td>
<td>Number of iterations (seeds). Use &quot;-n 0&quot; to run until fully routed and a timing score of zero is achieved. Default: 1.</td>
</tr>
<tr>
<td>-t</td>
<td>Start at this placer cost table entry. Default is 1.</td>
</tr>
<tr>
<td>-stopzero</td>
<td>Stop running iterations once a timing score of zero is achieved.</td>
</tr>
<tr>
<td>-s</td>
<td>Save &quot;n&quot; best results for this run. Default: Save All.</td>
</tr>
<tr>
<td>-m</td>
<td>Multi task par run. File &quot;&lt;node list file&gt;&quot;, contains a list of node names to run the jobs on.</td>
</tr>
<tr>
<td>-cores</td>
<td>Run multiple threads on the local machine. You can specify &quot;&lt;number of cores&gt;&quot; to run the jobs. For cases when the user specifies both -cores and -m with a valid node list file, PAR should apply both settings (merge). If the user repeats the host machine in the node list file, the settings in the node list file take precedence over the setting in -cores (for backwards compatibility).</td>
</tr>
<tr>
<td>-p</td>
<td>Don't run placement.</td>
</tr>
<tr>
<td>-r</td>
<td>Don't run router.</td>
</tr>
<tr>
<td>-k</td>
<td>Keep existing routing in input UDB file.</td>
</tr>
<tr>
<td>-x</td>
<td>Ignore timing constraints.</td>
</tr>
<tr>
<td>-pack</td>
<td>Set the packing density parameter. Default: auto.</td>
</tr>
</tbody>
</table>
Examples  Following are a few examples of PAR command lines and a description of what each does.

Example 1  The following command places and routes the design in the file input.udb and writes the placed and routed design to output.udb.

    par input.udb output.udb

Example 2  The following command runs 20 place and route iterations. The iterations begin at cost table entry 5. Only the best 3 output design files are saved.

    par -n 20 -t 5 -s 3 input.udb output.udb

Example 3  (Lattice FPGAs only) This is an example of par using the -io switch to generate .udb files that contain only I/O for viewing in the PAD Specification file for adjustment of ldc_set_location constraints for optimal I/O placement. You can display I/O placement assignments in the Radiant Spreadsheet View and choosing View > Display IO Placement.

    par -io -w lev1bist.udb lev1bist_io.udb
Using the PAR Multi-Tasking (-m) Option  This section provides information about environment setup, node list file creation, and step-by-step instructions for running the PAR Multi-tasking (-m) option from the command line. The PAR -m option allows you to use multiple machines (nodes) that are networked together for a multi-run PAR job, significantly reducing the total amount of time for completion. Before the multi-tasking option was developed, PAR could only run multiple jobs in a linear or serial fashion. The total time required to complete PAR was equal to the amount of time it took for each of the PAR jobs to run.

For example, the PAR command:

```
par -n 10 mydesign.udb output.udb
```

tells PAR to run 10 place and route passes (-n 10). It runs each of the 10 jobs consecutively, generating an output .udb file for each job, i.e., output_par.dir/5_1.udb, output_par.dir/5_2.udb, etc. If each job takes approximately one hour, then the run takes approximately 10 hours.

Suppose, however, that you have five nodes available. The PAR Multi-tasking option allows you to use all five nodes at the same time, dramatically reducing the time required for all ten jobs.

To run the PAR multi-tasking option from the command line:

1. First generate a file containing a list of the node names, one per line as in the following example:

```
# This file contains a profile node listing for a PAR multi
# tasking job.
[machine1]
SYSTEM = linux
CORENUM = 2
[machine2]
SYSTEM = linux
CORENUM = 2
Env = /home/user/setup_multipar.lin
Workdir = /home/user/myworkdir
```

You must use the format above for the node list file and fill in all required parameters. Parameters are case insensitive. The node or machine names are given in square brackets on a single line.

The **System** parameter can take linux or pc values depending upon your platform. However, the PC value cannot be used with Linux because it is not possible to create a multiple computer farm with PCs. **Corenum** refers to the number of CPU cores available. Setting it to zero will disable the node from being used. Setting it to a greater number than the actual number of CPUs will cause PAR to run jobs on the same CPU lengthening the runtime.

The **Env** parameter refers to a remote environment setup file to be executed before PAR is started on the remote machine. This is optional. If the remote machine is already configured with the proper environment, this line can be omitted. To test to see if the remote environment is responsive to PAR commands, run the following:
ssh <remote_machine> par <par_option>

See the System Requirements section below for details on this parameter.

**Workdir** is the absolute path to the physical working directory location on the remote machine where PAR should be run. This item is also optional. If an account automatically changes to the proper directory after login, this line can be omitted. To test the remote directory, run the following,

```bash
ssh <remote_machine> ls <udb_file>
```

If the design can be found then the current directory is already available.

2. Now run the job from the command line as follows:

```bash
par -m nodefile_name -n 10 mydesign.udb output.udb
```

This runs the following jobs on the nodes specified.

```
Starting job 5_1 on node NODE1 at ... 
Starting job 5_2 on node NODE2 at ... 
Starting job 5_3 on node NODE3 at ... 
Starting job 5_4 on node NODE4 at ... 
Starting job 5_5 on node NODE5 at ...
```

As the jobs finish, the remaining jobs start on the nodes until all 10 jobs are complete. Since each job takes approximately one hour, all 10 jobs will complete in approximately two hours.

**Note**

If you attempt to use the multi-tasking option and you have specified only one placement iteration, PAR will disregard the `-m` option from the command and run the job in normal PAR mode. In this case you will see the following message:

```
WARNING - par: Multi task par not needed for this job. -m switch will be ignored.
```

**System Requirements**  *ssh* must be located through the PATH variable. On Linux, the utility program’s secure shell (*ssh*) and secure shell daemon (*sshd*) are used to spawn and listen for the job requests.

The executables required on the machines defined in the node list file are as follows:

- `/bin/sh`
- `par` (must be located through the PATH variable)

Required environment variable on local and remote machines are as follows:

- `FOUNDRY` (points at FOUNDRY directory structure must be a path accessible to both the machine from which the Implementation Engine is run and the node)
- `LM_LICENSE_FILE` (points to the security license server nodes)
LD_LIBRARY_PATH (supports par path for shared libraries must be a path accessible to both the machine from which the Implementation Engine is run and the node)

To determine if everything is set up correctly, you can run the ssh command to the nodes to be used.

Type the following:

```
ssh <machine_name> /bin/sh -c par
```

If you get the usage message back on your screen, everything is set correctly. Note that depending upon your setup, this check may not work even though your status is fine.

If you have to set up your remote environment with the proper environment variables, you must create a remote shell environment setup file. An example of an ASCII file used to setup the remote shell environment would be as follows for ksh users:

```
export FOUNDRY=<install_directory>/ispfpga/bin/lin64
export PATH=$FOUNDRY/bin/lin64:$PATH
export LD_LIBRARY_PATH=$FOUNDRY/bin/lin:$LD_LIBRARY_PATH
```

For csh users, you would use the setenv command.

**Screen Output** When PAR is running multiple jobs and is not in multi-tasking mode, output from PAR is displayed on the screen as the jobs run. When PAR is running multiple jobs in multi-tasking mode, you only see information regarding the current status of the feature.

For example, when the job above is executed, the following screen output would be generated:

```
Starting job 5_1 on node NODE1
Starting job 5_2 on node NODE2
Starting job 5_3 on node NODE3
Starting job 5_4 on node NODE4
Starting job 5_5 on node NODE5
```

When one of the jobs finishes, this message will appear:

```
Finished job 5_3 on node NODE3
```

These messages continue until there are no jobs left to run.

**See Also**

- “Implementing the Design” in the Radiant software online help
- “Command Line Data Flow” on page 101
- “Command Line Program Overview” on page 99
Running Timing from the Command Line

The MAP Timing and Place & Route Timing processes in the Radiant software environment can also be run through the command line using the timing program. Timing can be run on designs using the placed and routed Unified Design Database (.udb) and associated timing constraints specified in the design's (.ldc,.fdc,.sdc or .pdc) file or device constraints extracted from the design. Using these input files, timing provides static timing analysis and outputs a timing report file (.tw1/.twr).

Timing checks the delays in the Unified Design Database (.udb) file against your timing constraints. If delays are exceeded, Timing issues the appropriate timing error. See “Implementing the Design” in the Radiant software online help and associated topics for more information.

Subjects included in this topic:

- Running Timing
- Command Line Syntax
- Timing Options
- Examples

Running Timing

Timing uses your input mapped or placed-and-routed Unified Design Database (.udb) file and associated constraint file to create a Timing Report.

To run Timing, type timing on the command line with, at minimum, the names of your input .udb and sdc files to output a timing report (.twr) file. A sample of a typical Timing command would be as follows:

```
timing design.udb (constraint is embedded in udb)
```

Note

The above command automatically generates the report file named design.twr which is based on the name of the .udb file.

There are several command line options that give you control over the way Timing generates timing reports for analysis. Please refer to the rest of the subjects in this topic for more details. See “Examples” on page 106.

Command Line Syntax

```
```

Lattice Radiant Software 2.0 User Guide 131
Timing Options  The following tables contain descriptions of all valid options for Timing.

Table 11: Compulsory Timing Command Line Options

<table>
<thead>
<tr>
<th>Compulsory Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-db-file arg</td>
<td>design database file name.</td>
</tr>
</tbody>
</table>

Table 12: Optional Timing Command Line Options

<table>
<thead>
<tr>
<th>Optional Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-endpoints arg (=10)</td>
<td>number of end points.</td>
</tr>
<tr>
<td>-u arg (=10)</td>
<td>number of unconstrained end points printed in the table.</td>
</tr>
<tr>
<td>-ports arg (=10)</td>
<td>number of top ports printed in the table.</td>
</tr>
<tr>
<td>-help</td>
<td>print the usage and exit.</td>
</tr>
<tr>
<td>-hld</td>
<td>hold report only.</td>
</tr>
<tr>
<td>-sp arg (=None)</td>
<td>Setup speed grade.</td>
</tr>
<tr>
<td>-hsp arg (=M)</td>
<td>Hold speed grade.</td>
</tr>
<tr>
<td>-rpt-file arg</td>
<td>timing report file name.</td>
</tr>
<tr>
<td>-o arg</td>
<td>timing report file name.</td>
</tr>
<tr>
<td>-alt_report</td>
<td>Diamond like report.</td>
</tr>
<tr>
<td>-sdc-file arg</td>
<td>sdc file name.</td>
</tr>
<tr>
<td>-sethld</td>
<td>both setup and hold report.</td>
</tr>
<tr>
<td>-v arg (=10)</td>
<td>number of paths per constraint.</td>
</tr>
<tr>
<td>-time_through_async</td>
<td>Timer will time through async resets.</td>
</tr>
<tr>
<td>-iotime</td>
<td>compute the input setup/hold and clock to output delays of the FPGA.</td>
</tr>
<tr>
<td>-io_allspeed</td>
<td>Get worst IO results for all speed grades.</td>
</tr>
<tr>
<td>-pwrprd</td>
<td>Output clock information for PowerCalculator.</td>
</tr>
<tr>
<td>-nperend arg (=1)</td>
<td>Number of paths per end point.</td>
</tr>
<tr>
<td>-html</td>
<td>HTML format report.</td>
</tr>
<tr>
<td>-gui</td>
<td>Call from GUI.</td>
</tr>
<tr>
<td>-msg arg</td>
<td>Message log file.</td>
</tr>
<tr>
<td>-msgset arg</td>
<td>Message setting.</td>
</tr>
</tbody>
</table>

Examples  Following are a few examples of Timing command lines and a description of what each does.
Example 1  The following command verifies the timing characteristics of the design named design1.udb, generating a summary timing report. Timing constraints contained in the file group1.prf are the timing constraints for the design. This generates the report file design1.twr.

timing design1.udb (constraint is embedded in udb)

Example 2  The following command produces a file listing all delay characteristics for the design named design1.udb. Timing constraints contained in the file group1.prf are the timing constraints for the design. The file output.twr is the name of the verbose report file.

timing -v design1.udb -o output.twr

Example 3  The following command analyzes the file design1.udb and reports on the three worst errors for each constraint in timing.prf. The report is called design1.twr.

timing -e 3 design1.udb

Example 4  The following command analyzes the file design1.udb and produces a verbose report to check on hold times on any FREQUENCY, CLOCK_TO_OUT, INPUT_SETUP and OFFSET constraints in the timing.prf file. With the output report file name unspecified here, a file using the root name of the .udb file (i.e., design1.twr) will be output by default.

timing -v -hld design1.udb

Example 5  The following command analyzes the file design1.udb and produces a summary timing report to check on both setup and hold times on any INPUT_SETUP and CLOCK_TO_OUT timing constraints in the timing.prf file. With the output report file name unspecified here, a file using the root name of the .udb file (i.e., design1.twr) will be output by default.

timing -sethld design1.udb

See Also
- “Command Line Program Overview” on page 99
- “Command Line Data Flow” on page 101

Running Backannotation from the Command Line

The Generate Timing Simulation Files process in the Radiant software environment can also be run through the command line using the backanno program. The backanno program back-annotates physical information (e.g., net delays) to the logical design and then writes out the back-annotated design in the desired netlist format. Input to backanno is a Unified Database file (.udb) a mapped and partially or fully placed and/or routed design.

Subjects included in this topic:
RUNNING BACKANNO

backanno uses your input mapped and at least partially placed-and-routed Unified Database (.udb) file to produce a back-annotated netlist (.v) and standard delay (.sdf) file. This tool supports all FPGA design architecture flows. Only Verilog netlist is generated.

To run backanno, type `backanno` on the command line with, at minimum, the name of your input .udb file. A sample of a typical backanno command would be as follows:

```
backanno backanno.udb
```

**Running Backanno**

There are several command line options that give you control over the way backanno generates back-annotated netlists for simulation. Please refer to the rest of the subjects in this topic for more details.

**Command Line Syntax (Verilog)**

```
```

**Backanno Options**

The table below contains descriptions of all valid options for backanno.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-w</code></td>
<td>Overwrite the output files.</td>
</tr>
<tr>
<td><code>-sp &lt;grade&gt;</code></td>
<td>Override performance grade for backannotation.</td>
</tr>
<tr>
<td><code>-pre &lt;prfx&gt;</code></td>
<td>Prefix to add to module name to make them unique for multi-chip simulation.</td>
</tr>
<tr>
<td><code>-min</code></td>
<td>Override performance grade to minimum timing for hold check.</td>
</tr>
<tr>
<td><code>-dis &lt;del&gt;</code></td>
<td>Distribute routing delays by splitting the signal and inserting buffers. &lt;del&gt; is the maximum delay (in ps) between each buffer (1000ps by default).</td>
</tr>
<tr>
<td><code>-m &lt;limit&gt;</code></td>
<td>Shortens the block names to a given character limit in terms of some numerical integer value.</td>
</tr>
</tbody>
</table>
Examples  Following are a few examples of backanno command lines and a description of what each does.

Example 1  The following command back annotates design.udb and generates a Verilog file design.vo and an SDF file design.sdf. If the target files exist, they will be overwritten.

```
backanno -w design.udb
```

Example 2  The following command back annotates design.udb and generates a Verilog file backanno.vo and an SDF file backanno.sdf. Any signal in the design that has an interconnection delay greater than 2000 ps (2 ns) will be split and a series of buffers will be inserted. The maximum interconnection delay between each buffer would be 2000 ps.

```
backanno -dis 2000 -o backanno design.udb
```

Example 3  The following command re-targets backannotation to peforcxmance grade -2, and puts a buffer at each block input to isolate the interconnection delay (ends at that input) and the pin to pin delay (starts from that input).

```
backanno -sp 2 -i design.udb
```

Example 4  The following command generates Verilog netlist and SDF files without setting the negative setup/hold delays to 0:

```
-u
-neg
-pos
-x
-i
-nopur
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-u</td>
<td>Add pads for top-level dangling nets.</td>
</tr>
<tr>
<td>-neg</td>
<td>Negative setup/hold delay support. Without this option, all negative numbers are set to 0 in SDF.</td>
</tr>
<tr>
<td>-pos</td>
<td>Write out 0 for negative setup/hold time in SDF for SC.</td>
</tr>
<tr>
<td>-x</td>
<td>Generate x for setup/hold timing violation.</td>
</tr>
<tr>
<td>-i</td>
<td>Create a buffer for each block input that has interconnection delay.</td>
</tr>
<tr>
<td>-nopur</td>
<td>Do not write PUR instance in the backannotation netlist. Instead, user has to instantiate it in a test bench.</td>
</tr>
<tr>
<td>&lt;type&gt;</td>
<td>Netlist type to write out.</td>
</tr>
<tr>
<td>&lt;libtype&gt;</td>
<td>Library element type to use.</td>
</tr>
<tr>
<td>&lt;netfile&gt;</td>
<td>The name of the output netlist file. The extension on this file will change depending on which type of netlist is being written. Use -h &lt;type&gt;, where &lt;type&gt; is the output netlist type, for more specific information.</td>
</tr>
<tr>
<td>&lt;udb file&gt;</td>
<td>Input file '.udb '.</td>
</tr>
</tbody>
</table>
Running Bit Generation from the Command Line

The Bitstream process in the Radiant software environment can also be run through the command line using the bit generation (bitgen) program. This topic provides syntax and option descriptions for usage of the bitgen program from the command line. The bitgen program takes a fully routed Unified Database (..udb) file as input and produces a configuration bitstream (bit images) needed for programming the target device.

Subjects included in this topic:
- Running BITGEN
- Command Line Syntax
- BITGEN Options
- Examples

Running BITGEN  BITGEN uses your input, fully placed-and-routed Unified Database (.udb) file to produce bitstream (.bit, .msk, or .rbt) for device configuration.

To run BITGEN, type bitgen on the command line with, at minimum, the bitgen command. There is no need to specify the input .udb file if you run bitgen from the directory where it resides and there is no other .udb present.

There are several command line options that give you control over the way BITGEN outputs bitstream for device configuration. Please refer to the rest of the subjects in this topic for more details.


BITGEN Options  The table below contains descriptions of all valid options for BITGEN.

Note  Many BITGEN options are only available for certain architectures. Please use the bitgen -h <architecture> help command to see a list of valid bitgen options for the particular device architecture you are targeting.
### Table 14: BITGEN Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d</td>
<td>Disable DRC.</td>
</tr>
<tr>
<td>-b</td>
<td>Produce .rbt file (ASCII form of binary).</td>
</tr>
<tr>
<td>-a</td>
<td>Produce .hex file.</td>
</tr>
<tr>
<td>-w</td>
<td>Overwrite an existing output file.</td>
</tr>
<tr>
<td>-freq</td>
<td>Can setup different frequency: 0 = slow, 1 = medium, 2 = fast.</td>
</tr>
<tr>
<td>&lt;frequency_bit_setting&gt;</td>
<td>Depending on the speed of external PROM, this options adjusts the frequency of the internal oscillator used by the iCE40UP device during configuration. This is only applicable when the iCE40UP device is used in SPI Master Mode for configuration.</td>
</tr>
<tr>
<td>-nvcm</td>
<td>Produce NVCM file.</td>
</tr>
<tr>
<td>-nvcmsecurity</td>
<td>Set security. Ensures that the contents of the Non-Volatile Configuration Memory (NVCM) are secure and the configuration data cannot be read out of the device.</td>
</tr>
<tr>
<td>-spilowpower</td>
<td>SPI flash low power mode. Places the PROM in low-power mode after configuration.</td>
</tr>
<tr>
<td></td>
<td>This option is applicable only when the iCE40UP device is used as SPI Master Mode for configuration.</td>
</tr>
<tr>
<td>-warmboot</td>
<td>Enable warm boot. Enables the Warm Boot functionality, provided the design contains an instance of the WARMBOOT primitive.</td>
</tr>
<tr>
<td>-noheader</td>
<td>Don’t include the bitstream header.</td>
</tr>
<tr>
<td>-noebrinitq0</td>
<td>Don’t include EBR initialization for quadrant 0.</td>
</tr>
<tr>
<td>-noebrinitq1</td>
<td>Don’t include EBR initialization for quadrant 1.</td>
</tr>
<tr>
<td>-noebrinitq2</td>
<td>Don’t include EBR initialization for quadrant 2.</td>
</tr>
<tr>
<td>-noebrinitq3</td>
<td>Don’t include EBR initialization for quadrant 3.</td>
</tr>
<tr>
<td>-g NOPULLUP:ENABLED</td>
<td>No IO pullup. Removes the pullup on the unused I/Os, except Bank 3 I/Os which do not have pullup.</td>
</tr>
<tr>
<td>-h &lt;architecture&gt; or -help &lt;architecture&gt;</td>
<td>Display available BITGEN command options for the specified architecture. The bitgen -h command with no architecture specified will display a list of valid architectures.</td>
</tr>
</tbody>
</table>
Example  The following command tells bitgen to overwrite any existing bitstream files with the -w option, prevents a physical design rule check (DRC) from running with -d, specifies a raw bits (.rbt) file output with -b. Notice how these three options can be combined with the -wdb syntax.

\[\text{bitgen} \ -wdb\ \text{PERSIST:Yes}\]

See Also  
- “Command Line Program Overview” on page 99
- “Command Line Data Flow” on page 101

Running Programmer from the Command Line
You can run Programmer from the command line. The PGRCMD command uses a keyword preceded by a hyphen for each command line option.

Running PGRCMD  PGRCMD allows you to download data files to an FPGA device.

To run PGRCMD, type \text{pgrcmd} on the command line with, at minimum, the \text{pgrcmd} command.

There are several command line options that give you control over the way PGRCMD programs devices. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax  The following describes the PGRCMD command line syntax:

\text{pgrcmd [-help] [-infile <input_file_path>] [-logfile <log_file_path>] [-cabletype <cable>]

- \text{-cabletype}
  - lattice [ -portaddress < 0x0378 | 0x0278 | 0x03bc | 0x<custom address> > ]
  - usb [ -portaddress < EZUSB-0 | EZUSB-1 | ... | EZUSB-15 > ]
  - usb2 [ -portaddress < FTUSB-0 | FTUSB-1 | ... | FTUSB-15 > ]
TCK [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**PGRCMD Options**  The following are PGRCMD options.

**Help (Optional)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help or -h</td>
<td>Displays the Programmer command line options.</td>
</tr>
</tbody>
</table>

**Input File (required)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>infile filename.xcf</td>
<td>Specifies the chain configuration file (.xcf). If the file path includes spaces, enclose the path in quotes.</td>
</tr>
</tbody>
</table>

**Log File (optional)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logfile logfilename.log</td>
<td>Specifies the location of the Programmer log file.</td>
</tr>
</tbody>
</table>

**Cable Type (optional)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cabletype lattice</td>
<td>Lattice HW-DLN-3C parallel port programming cable (default).</td>
</tr>
<tr>
<td>cabletype usb</td>
<td>Lattice HW-USBN-2A USB port programming cable.</td>
</tr>
<tr>
<td>cabletype usb2</td>
<td>Lattice FHW-USBN-2B (FTDI) USB programming cable and any FTDI based demo boards.</td>
</tr>
</tbody>
</table>

**Parallel Port Address (optional)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>portaddress 0x0378</td>
<td>LPT1 parallel port (default)</td>
</tr>
<tr>
<td>portaddress 0x0278</td>
<td>LPT2 parallel port</td>
</tr>
<tr>
<td>portaddress 0x03BC</td>
<td>LPT3 parallel port</td>
</tr>
<tr>
<td>portaddress 0x&lt;custom address&gt;</td>
<td>Custom parallel port address</td>
</tr>
</tbody>
</table>

This option is only valid with parallel port cables.
USB Port Address (optional)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-portaddress EZUSB-0 ... EZUSB-15</td>
<td>HW-USBN-2A USB cable number 0 through 15</td>
</tr>
<tr>
<td>-portaddress FTUSB-0 ... FTUSB-15</td>
<td>FTDI based demo board or FTDI USB2 cable number 0 through 15</td>
</tr>
</tbody>
</table>

Default is EZUSB-0 and FTUSB-0. Only valid with the USB port cables.

FTDI Based Demo Board or Cable Frequency Control (optional)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-TCK 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10</td>
<td>0 = 30 Mhz 1 = 15 Mhz (default) 2 = 10 Mhz 3 = 7.5 Mhz 4 = 6 Mhz 5 = 5 Mhz 6 = 4 Mhz 7 = 3 Mhz 8 = 2 Mhz 9 = 1 Mhz 10 = 900 Khz</td>
</tr>
</tbody>
</table>

Calculation formula for USB-2B (2232H FTDI USB host chip): Frequency = 60 MHz / (1 + ClockDivider) *2

Calculation formula for USB-2B (2232D FTDI USB host chip): Frequency = 12 MHz / (1 + ClockDivider) *2

Only applicable for FTDI based demo boards or programming cable.

Return Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Log file error</td>
</tr>
<tr>
<td>-2</td>
<td>Check configuration setup error</td>
</tr>
<tr>
<td>-3</td>
<td>Out of memory error</td>
</tr>
<tr>
<td>-4</td>
<td>NT driver error</td>
</tr>
</tbody>
</table>
### Code Definition

<table>
<thead>
<tr>
<th>Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>Cable not detected error</td>
</tr>
<tr>
<td>-6</td>
<td>Power detection error</td>
</tr>
<tr>
<td>-7</td>
<td>Device not valid error</td>
</tr>
<tr>
<td>-8</td>
<td>File not found error</td>
</tr>
<tr>
<td>-9</td>
<td>File not valid error</td>
</tr>
<tr>
<td>-10</td>
<td>Output file error</td>
</tr>
<tr>
<td>-11</td>
<td>Verification error</td>
</tr>
<tr>
<td>-12</td>
<td>Unsupported operation error</td>
</tr>
<tr>
<td>-13</td>
<td>File name error</td>
</tr>
<tr>
<td>-14</td>
<td>File read error</td>
</tr>
<tr>
<td>-17</td>
<td>Build SVF file error</td>
</tr>
<tr>
<td>-18</td>
<td>Build VME file error</td>
</tr>
<tr>
<td>-19</td>
<td>Command line syntax error</td>
</tr>
</tbody>
</table>

### Examples

The following is a PGRCMD example.

```
pgrcmd -infile \c:\test.xcf
```

### See Also

- “Command Line Data Flow” on page 101
- “Command Line Program Overview” on page 99

### Running Various Utilities from the Command Line

The command line utilities described in this section are not commonly used by command line users, but you often see them in the auto-make log when you run design processes in the Radiant software environment. Click each link below for its function, syntax, and options.

### Note

For information on commonly-used FPGA command line tools, see “Command Line Basics” on page 101.

### Synpwrap

The `synpwrap` command line utility (wrapper) is used to manage Synplicity Synplify and Synplify Pro synthesis programs from the Radiant software environment processes: Synplify Synthesize Verilog File or Synplify Synthesize VHDL File.
The `synpwrap` utility can also be run from the command line to support a batch interface. For details on Synplify see the Radiant software online help. The `synpwrap` program drives `synplify_pro` programs with a Tcl script file containing the synthesis options and file list.

**Note**
This section supersedes the “Process Optimization and Automation” section of the Synplicity Synplify and Synplify Pro for Lattice User Guide.

This section illustrates the use of the `synpwrap` program to run Synplify Pro for Lattice synthesis scripts from the command line. For more information on synthesis automation of Synplify Pro, see the “User Batch Mode” section of the Synplicity Synplify and Synplify Pro for Lattice User Guide.

If you use Synplify Pro, the Lattice OEM license requires that the command line executables `synplify_pro` be run by the Lattice “wrapper” program, `synpwrap`.

**Command Line Syntax**

```
```

**Table 15: SYNPWRAP Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-log &lt;log_file&gt;</code></td>
<td>Specifies the log file name.</td>
</tr>
<tr>
<td><code>-nolog</code></td>
<td>Does not print out the log file after the process is finished.</td>
</tr>
<tr>
<td><code>-options &lt;arguments&gt;</code></td>
<td>Passes all arguments to Synplify/Pro. Ignores all other options except <code>-notoem/-oem</code> and <code>-notpro/-pro</code>. The <code>-options</code> switch must follow all other <code>synpwrap</code> options.</td>
</tr>
<tr>
<td><code>-prj &lt;project_file&gt;</code></td>
<td>Runs Synplify or Synplify Pro using an external prj Tcl file instead of the Radiant software command file.</td>
</tr>
<tr>
<td><code>-rem</code></td>
<td>Does not automatically include Lattice library files.</td>
</tr>
<tr>
<td><code>-e &lt;command_file&gt;</code></td>
<td>Runs the batch interface based on a Radiant software generated command file. The <code>synpwrap</code> utility reads <code>&lt;project&gt;.cmd</code> with its command line to obtain user options and creates a Tcl script file.</td>
</tr>
<tr>
<td><code>-gui</code></td>
<td>Invokes the Synplify or Synplify Pro graphic user interface.</td>
</tr>
<tr>
<td><code>-int &lt;command_file&gt;</code></td>
<td>Enables the interactive mode. Runs Synplify/Pro UI with project per command file.</td>
</tr>
<tr>
<td><code>-dyn</code></td>
<td>Brings the Synplify installation settings in the Radiant software environment.</td>
</tr>
<tr>
<td><code>-notoem</code></td>
<td>Does not use the Lattice OEM version of Synplify or Synplify Pro.</td>
</tr>
</tbody>
</table>
Table 15: SYNPWRAP Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-oem</td>
<td>Uses the Lattice OEM version of Synplify or Synplify Pro.</td>
</tr>
<tr>
<td>-notpro</td>
<td>Does not use the Synplify Pro version.</td>
</tr>
<tr>
<td>-pro</td>
<td>Uses the Synplify Pro version.</td>
</tr>
<tr>
<td>-target &lt;device_family&gt;</td>
<td>Specifies the device family name.</td>
</tr>
<tr>
<td>-part &lt;device_name&gt;</td>
<td>Specifies the device. For details on legal &lt;device_name&gt; values.</td>
</tr>
<tr>
<td>-scriptonly &lt;script_file&gt;</td>
<td>Generates the Tcl file for Synplify or Synplify Pro. Does not run synthesis.</td>
</tr>
</tbody>
</table>

Example
Below shows a synpwrap command line example.
synpwrap -rem -e prep1 -target iCE40UP

See Also
► “Command Line Program Overview” on page 99
► “Command Line Data Flow” on page 101

IP Packager
The IP Packager (ippkg) tool can be run from the command line, allowing IP developers to select files from disks and pack them into one IPK file.

The process of IP packager is as following:
► IP author prepares metadata files, RTL files, HTML files, etc (all files of a Soft IP).
► IP Packager GUI provides UI for IP author to select files from the disk, and call IP Packaging engine to pack them into an IPK file.
► IP Packaging engine encrypts RTL files if IEEE P1735-2014 V1 pragmas are specified in RTL source

Command Line Syntax


Table 16: IPPKG Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name</td>
<td>Specify the IP name.</td>
</tr>
<tr>
<td>-metadata</td>
<td>The file name will be fixed to ‘metadata.xml’.</td>
</tr>
</tbody>
</table>
Table 16: IPPKG Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-metadata_files</td>
<td>Location of the file which stores the metadata files. One line is a file path in specified file. Must have a file named metadata.xml.</td>
</tr>
<tr>
<td>-rtl</td>
<td>Specify the IP RTL file.</td>
</tr>
<tr>
<td>-rtl_files</td>
<td>One line is a file path in specified file.</td>
</tr>
<tr>
<td>-plugin</td>
<td>The file name will be fixed to 'plugin.py'.</td>
</tr>
<tr>
<td>-ldc</td>
<td>Specify the LDC file.</td>
</tr>
<tr>
<td>-testbench</td>
<td>Specify the testbench file.</td>
</tr>
<tr>
<td>-testbench_files</td>
<td>One line is a file path in specified file.</td>
</tr>
<tr>
<td>-help_file</td>
<td>Specify the help file, must be &lt;path&gt;/introduction.html.</td>
</tr>
<tr>
<td>-help_files</td>
<td>One line is a file path in specified file.</td>
</tr>
<tr>
<td>-license_file</td>
<td>Specify the license file.</td>
</tr>
<tr>
<td>-o</td>
<td>Specify the output zip file.</td>
</tr>
<tr>
<td>-key_file</td>
<td>Specify the key file to encrypt the RTL files.</td>
</tr>
<tr>
<td>--force-run</td>
<td>Force program to run regardless of errors.</td>
</tr>
</tbody>
</table>

Example  The following is an ippkg command line example:

```
ippkg -metadata c:/test/test.xml -rtl_files c:/test/rtl_list -help_file c:/test/introduction.html
```

See Also  ►“Command Line Program Overview” on page 99
► “Command Line Data Flow” on page 101

ECO Editor

The ECO Editor tool can be run from the command line too.

ECO Editor is also able to dump the ECO TCL commands which user acted in GUI view without saving any UDB file.

In the meanwhile, we will have one non-GUI ECO engine tool, it accepts the dumped TCL script file with a UDB file and output a new UDB file.

User can set ‘Place & Route design’ milestone post-script by Tcl command prj_set_postscript par <eco.tcl>, then Radiant flow runs the ECO Tcl script automatically after running place & route.
**Command Line Syntax**

```plaintext
ecoc [-s <script_file>] [-o <output.udb>] <input.udb>
```

**Table 17: ECO Editor Command Line Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s</td>
<td>ECO Tcl script file.</td>
</tr>
<tr>
<td>-o</td>
<td>Output UDB file.</td>
</tr>
<tr>
<td>&lt;input.udb&gt;</td>
<td>Input UDB file.</td>
</tr>
</tbody>
</table>

**Example**

The following is an ecoc command line example:

```plaintext
ecoc -s mem.tcl ebr_test_impl_1.udb
```

**See Also**

- "Command Line Program Overview" on page 99
- "Command Line Data Flow" on page 101

---

**Using Command Files**

This section describes how to use command files.

**Creating Command Files**

The command file is an ASCII file containing command arguments, comments, and input/output file names. You can use any text editing tool to create or edit a command file, for example, **vi**, **emacs**, **Notepad**, or **Wordpad**.

Here are some guidelines when you should observe when creating command files:

- Arguments (executables and options) are separated by space and can be spread across one or more lines within the file.
- Place new lines or tabs anywhere white space would otherwise be allowed on the Linux or DOS command line.
- Place all arguments on the same line, or one argument per line, or any combination of the two.
- There is no line length limitation within the file.
- All carriage returns and other non-printable characters are treated as space and ignored.
- Comments should be preceded with a # (pound sign) and go to the end of the line.

**Command File Example**

This is an example of a command file:

```plaintext
#command line options for par for design mine.udb
-a -n 10
-w
-l 5
-s 2 #will save the two best results
/home/users/jimbob/designs/mine.udb
```
Using the Command File  The –f Option  Use the –f option to execute a command file from any command line tool. The –f option allows you to specify the name of a command file that stores and then executes commonly used or extensive command arguments for a given FPGA command line executable tool. You can then execute these arguments at any time by entering the Linux or DOS command line followed by the name of the file containing the arguments. This can be useful if you frequently execute the same arguments each time you perform the command, or if the command line becomes too long. This is the recommended way to get around the DOS command line length limitation of 127 characters. (Equivalent to specifying a shell Options file.)

The –f indicates fast startup, which is performed by not reading or executing the commands in your .cshrc | .kshrc | .shrc (C-shell, Korn-shell, Bourne-shell) file. This file typically contains your path information, your environment variable settings, and your aliases. By default, the system executes the commands in this file every time you start a shell. The –f option overrides this process, discarding the ‘set’ variables and aliases you do not need, making the process much faster. In the event you do need a few of them, you can add them to the command file script itself.

Command File Usage Examples  You can use the command file in two ways:

▶ To supply all of the command arguments as in this example:

par -f <command_file>

where:

<command_file> is the name of the file containing the command line arguments.

▶ To insert certain command line arguments within the command line as in the following example:

par -i 33 -f placeoptions -s 4 -f routeoptions design_i.udb design_o.udb

where:

placeoptions is the name of a file containing placement command arguments.

routeoptions is the name of a file containing routing command arguments.
Using Command Line Shell Scripts

This topic discusses the use of shell scripts to automate either parts of your design flow or entire design flows. It also provides some examples of what you can do with scripts. These scripts are Linux-based; however, it is also possible to create similar scripts called batch files for PC but syntax will vary in the DOS environment.

Creating Shell Scripts  A Linux shell script is an ASCII file containing commands targeted to a particular shell that interprets and executes the commands in the file. For example, you could target Bourne Shell (`sh`), C-Shell (`csh`), or Korn Shell (`ksh`). These files also can contain comment lines that describe part of the script which then are ignored by the shell. You can use any text editing tool to create or edit a shell script, for example, `vi` or `emacs`.

Here are some guidelines when you should observe when creating shell scripts:

- It is recommended that all shell scripts with "#!" followed by the path and name of the target shell on the first line, for example, `#!/bin/ksh`. This indicates the shell to be used to interpret the script.
- It is recommended to specify a search path because oftentimes a script will fail to execute for users that have a different or incomplete search path. For example:
  ```bash
  PATH=/home/usr/lsmith:/usr/bin:/bin; export PATH
  ```
- Arguments (executables and options) are separated by space and can be spread across one or more lines within the file.
- Place new lines or tabs anywhere white space would otherwise be allowed on the Linux command line.
- Place all arguments on the same line, or one argument per line, or any combination of the two.
- There is no line length limitation within the file.
- All carriage returns and other non-printable characters are treated as space and ignored.
- Comments are preceded by a `#` (pound sign) and can start anywhere on a line and continue until the end of the line.
- It is recommended to add exit status to your script, but this is not required.

```bash
# Does global timing meet acceptable requirement range?
if [ $timing -lt 5 -o $timing -gt 10 ]; then
  echo 1>&2 Timing "$timing" out of range
  exit 127
fi
etc...
# Completed, Exit OK
exit 0
```

Advantages of Using Shell Scripts  Using shell scripts can be advantageous in terms of saving time for tasks that are often used, in
reducing memory usage, giving you more control over how the FPGA design flow is run, and in some cases, improving performance.

**Scripting with DOS** Scripts for the PC are referred to as batch files in the DOS environment and the common practice is to ascribe a .bat file extension to these files. Just like Linux shell scripts, batch files are interpreted as a sequence of commands and executed. The COMMAND.COM or CMD.EXE (depending on OS) program executes these commands on a PC. Batch file commands and operators vary from their Linux counterparts. So, if you wish to convert a shell script to a DOS batch file or vice-versa, we suggest you find a good general reference that shows command syntax equivalents of both operating systems.

**Examples** The following example shows running design “counter” on below device package

- **Architecture:** iCE40UP
- **Device:** iCE40UP3K
- **Package:** UWG30
- **Performance:** Worst Case

**Command 1: logic synthesis**
```
synthesis -f counter_impl1_lattice.synproj
    which the *.synproj contains
    -a "iCE40UP"
    -p iCE40UP3K
    -t UWG30
    -sp "Worst Case"
    -optimization_goal Area
    -bram_utilization 100
    -ramstyle Auto
    -romstyle auto
    -dsp_utilization 100
    -use_dsp 1
    -use_carry_chain 1
    -carry_chain_length 0
    -force_gsr Auto
    -resource_sharing 1
    -propagate_constants 1
    -remove_duplicateRegs 1
    -mux_style Auto
    -max_fanout 1000
    -fsm_encoding_style Auto
    -twr_paths 3
    -fix_gated_clocks 1
    -loop_limit 1950
    -use_io_reg auto
    -use_io_insertion 1
    -resolve_mixed_drivers 0
    -sdc "impl1.ldc"
    -path "C:/lscc/radiant/1.0/ispfpga/ice40tp/data" "impl1"
    -ver "C:/lscc/radiant/1.0/ip/pmi/pmi.v"
    -ver "count_attr.v"
    -path "."
```
-top count
-udb "counter_impl1.udb"
-output_hdl "counter_impl1.vm"

**Command 2: post synthesis process**
postsyn -a iCE40UP -p iCE40UP3K -t UWG30 -sp Worst Case -top -
  ldc counter_impl1.ldc -keeprtl -w -o counter_impl1.udb
  counter_impl1.vm

**Command 3: Mapper**
map "counter_impl1_syn.udb" "impl1.pdc" -o "counter_impl1.udb"

**Command 4: Placer and router**
par -f "counter_impl1.p2t" "counter_impl1_map.udb"
"counter_impl1.udb"

**Command 5: Timer**
timing -sethld -v 10 -u 10 -endpoints 10 -nperend 1 -html -rpt
  "counter_impl1_trw.html" "counter_impl1.udb"

**Command 6: back annotation**
backanno "counter_impl1.udb" -n Verilog -o
  "counter_impl1.vo.vo" -w -neg

**Command 7: bitstream generation**
bitgen -w "counter_impl1.udb" -f "counter_impl1.t2b"
The Radiant software supports Tcl (Tool Command Language) scripting and provides extended Radiant software Tcl commands that enable a batch capability for running tools in the Radiant software’s graphical interface. The command set and the Tcl Console used to run it affords you the speed, flexibility and power to extend the range of useful tasks that the Radiant software tools are already designed to perform.

In addition to describing how to run the Radiant software’s Tcl Console, this guide provides you with a reference for Tcl command line usage and syntax for all Radiant software point tools within the graphical user interface so that you can create command scripts, modify commands, or troubleshoot existing scripts.

**About the Radiant software Tcl Scripting Environment**  The Radiant software development software features a powerful script language system. The user interface incorporates a complete Tcl command interpreter. The command interpreter is enhanced further with additional Radiant software-specific support commands. The combination of fundamental Tcl along with the commands specialized for use with the Radiant software allow the entire Radiant software development environment to be manipulated.

Using the command line tools permits you to do the following:

- Develop a repeatable design environment and design flow that eliminates setup errors that are common in GUI design flows
- Create test and verification scripts that allow designs to be checked for correct implementation
- Run jobs on demand automatically without user interaction

The Radiant software command interpreter provides an environment for managing your designs that are more abstract and easier to work with than using the core Radiant software engines. The Radiant software command interpreter does not prevent use of the underlying transformation tools. You
can use either the TCL commands described in this section or you can use the core engines described in the "Command Line Reference Guide" on page 99.

Additional References If you are unfamiliar with the Tcl language you can get help by visiting the Tcl/tk web site at http://www.tcl.tk. If you already know how to use Tcl, see the Tcl Manual supplied with this software. For information on command line syntax for running core tools that appear as Radiant software processes, such as synthesis, map, par, backanno, and timing, see the "Command Line Reference Guide" on page 99.

See Also ► "Running the Tcl Console" on page 151
► "Accessing Command Help in the Tcl Console" on page 153
► "Radiant Software Tool Tcl Command Syntax" on page 157
► "Creating and Running Custom Tcl Scripts" on page 153
► "Accessing Command Help in the Tcl Console" on page 153
► Tcl Manual (Windows only)
► Tcl Manual (Linux only)

Running the Tcl Console

The Radiant software TCL Console environment is made available for your use in multiple different ways. In order to take full advantage of the FPGA development process afforded by the Radiant software you must gain access to the Radiant Tcl Console user interface.

On Windows In Windows 7 you can interact with the Tcl Console by any one of the following methods:
► To launch the Radiant software GUI from the Windows Start menu, choose Start > All Programs > Lattice Radiant Software > Radiant Software.

After the the Radiant software loads you can click on the TCL Console tab. With the TCL Console tab active, you are able to start entering standard syntax TCL commands or the Radiant software specific support commands.

► To launch the TCL Console independently from the Radiant software GUI from the Windows Start menu choose Start > All Programs > Lattice Radiant Software > Accessories > TCL Console.

A Windows command interpreter will be launched that automatically runs the TCL Console.

► To run the interpreter from the command line, type the following:
c:/lscc/radiant/<version_number>/bin/nt64/pnmainc

The Radiant TCL Console is now available to run.
To run the interpreter from a Windows 7 PowerShell from the Windows Start menu choose Start > All Programs > Accessories > Windows PowerShell > Windows PowerShell (x86).

A PowerShell interpreter window will open. At the command line prompt type the following:

c:/lscc/radiant/<version_number>/bin/nt64/pnmainc

The Radiant Tcl Console is now available to run.

Note

The arrangement and location of each of the programs in the Windows Start menu will differ depending on the version of Windows you are running.

On Linux  In Linux operating systems you can interact with the Tcl Console by one of the following methods:

To launch the Radiant software GUI from the command line, type the following:

/usr/<user_name>/radiant/<version_number>/bin/lin64/radiant

The path provided assumes the default installation directory and that the Radiant software is installed. After the Radiant software loads you can click on the Tcl Console tab. With the Tcl Console tab active, you are able to start entering standard syntax TCL commands or the Radiant software specific support commands.

To launch the Tcl Console independently from the Radiant software GUI from the command line, type the following:

/usr/<user_name>/Radiant/<version_number>/bin/lin64/radiantc

The path provided assumes the default installation directory and that the Radiant software is installed, and that you have followed the Radiant software for Linux installation procedures. The Radiant Tcl Console is now ready to accept your input.

The advantage of running the Tcl Console from an independent command interpreter is the ability to directly pass the script you want to run to the Tcl interpreter. Another advantage is that you have full control over the Tk graphical environment, which allows you to create your own user interfaces.

See Also  ►“Running the Tcl Console” on page 151
► “Radiant Software Tool Tcl Command Syntax” on page 157
► “Creating and Running Custom Tcl Scripts” on page 153
► “Accessing Command Help in the Tcl Console” on page 153
► Tcl Manual (Windows only)
► Tcl Manual (Linux only)
Accessing Command Help in the Tcl Console

You can access command syntax help for all of the tools in the Tcl Console.

To access command syntax help in the Tcl Console:
1. In the prompt, type `help <tool_name>*` and press Enter as shown below:
   ```
   help prj*
   ```
   A list of valid command options appears in the Tcl Console.
2. In the Tcl Console, type the name of the command or function for more
details on syntax and usage. For the prj tool, for example, type and enter
the following:
   ```
   prj_open
   ```
   A list of valid arguments for that function appears.

Note

Although you can run the Radiant software’s core tools such as synthesis, postsyn,
map, par, and timing from the Tcl Console, the syntax for accessing help is different.
For proper usage and syntax for accessing help for core tools, see the “Command

See Also
- “Running the Tcl Console” on page 151
- “Radiant Software Tool Tcl Command Syntax” on page 157
- “Creating and Running Custom Tcl Scripts” on page 153
- “Running Tcl Scripts When Launching the Radiant Software” on page 156
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Creating and Running Custom Tcl Scripts

This topic describes how to easily create Tcl scripts using the Radiant
software’s user interface and manual methods. FPGA design using Tcl scripts
provides some distinct advantages over using the graphical user interface’s
lists, views and menu commands. For example, Tcl scripts allow you to do the
following:

- Set the tool environment to exactly the same state for every design run.
  This eliminates human errors caused by forgetting to manually set a
critical build parameter from a drop-down menu.
- Manipulate intermediate files automatically, and consistently on every run.
  For example, .vm file errors can be corrected prior to performing
  additional netlist transformation operations.
- Run your script automatically by using job control software. This gives you
  the flexibility to run jobs at any time of day or night, taking advantage of
  idle cycles on your corporate computer system.
Creating Tcl Scripts  There are a couple of different methods you can use to create the Radiant software Tcl scripts. This section will discuss each one and provide step-by-step instructions for you to get started Tcl scripting repetitious Radiant software commands or entire workflows.

One method you have available is to use your favorite text editor to enter a sequence of the Radiant software Tcl commands. The syntax of each the Radiant software Tcl commands is available in later topics in this portion of the online help. This method should only be used by very experienced Radiant software Tcl command line users.

The preferred method is to let the Radiant software GUI assist you in getting the correct syntax for each Tcl command. When you interact with the Radiant software user interface each time you launch a scriptable process and the corresponding Radiant software Tcl command is echoed to the Tcl Console. This makes it much simpler to get the correct command line syntax for each Radiant software command. Once you have the fundamental commands executed in the correct order, you can then add additional Tcl code to perform error checking, or customization steps.

To create a Tcl command script in the Radiant software:

1. Start the Radiant software design software and close any project that may be open.

2. In the Tcl Console execute the custom reset command. This clears the Tcl Console command history.

3. Use the Radiant software graphical user interface to start capturing the basic command sequence. The Tcl Console echos the commands in its window. Start by opening the project for which you wish to create the TCL script. Then click on the processes in the Process bar to run them. For example, run these processes in their chronological order in the design flow:
   - Synthesize Design
   - Map Design
   - Place & Route Design
   - Export Files

4. In the Tcl Console window enter the command,
   ```
   save_script <filename.ext>
   ```
   The <filename.ext> is any file identifier that has no spaces and contains no special characters except underscores. For example, myscript.tcl or design_flow_1.tcl are acceptable save_script values, but my$script or my script are invalid. The <filename.ext> entry can be preceded with a absolute or relative path. Use the "/" (i.e. forward slash) character to delimit the path elements. If the path is not specified explicitly the script is saved in the current working directory. The current working directory can be determined by using the TCL pwd command.

5. You can now use your favorite text editor to make any changes to the script you feel are necessary. Start your text editor, navigate to the
directory the save_script command saved the base script, and open the file.

Note
In most all cases, you will have to clean up the script you saved and remove any invalid arguments or any commands that cannot be performed in the Radiant software environment due to some conflict or exception. You will likely have to revisit this step later if after running your script you experience any run errors due to syntax errors or technology exceptions.

Sample Radiant software Tcl Script  The following the Radiant software Tcl script shows a very simple script that opens a project, runs the entire design flow through the Place & Route process, then closes the project. A typical script will contain more tasks and will check for failure conditions. Use this simple example as a general guideline.

Figure 92: Simple Radiant software Script

prj_archive -dir "C:/my_radiant/counter" -extract "C:/lscc/radiant/1.1/examples/counter.zip"
prj_run_par
prj_close

Running Tcl Scripts  The Radiant software TCL scripts are run exclusively from the Radiant TCL Console. You can use either the TCL Console integrated into the Radiant software UI, or by launching the stand-alone TCL Console.

To run a Tcl script in the Radiant software:

1. Launch the Radiant software GUI, or the stand-alone TCL Console.
   
Open the Radiant software but do not open your project. If your project is open, choose File > Close Project.

2. If you are using the Radiant software main window, click the small arrow pane switch in the bottom of the Radiant software main window, and then click on the Tcl Console tab in the Output area at the bottom to open the console.

3. Use the TCL source command to load and run your TCL script. The source command requires, as it's only argument, the filename of the script you want to load and run. Prefix the script file name with any required relative or absolute path information. To run the example script shown in the previous section use:

   source C:/lscc/radiant/<version_number>/examples/counter/myscript2.tcl

As long as there are no syntax errors or invalid arguments, the script will open the project, synthesize, map, and place-and-route the design. Once the design finishes it closes the project. If there are errors in the script, you will see the errors in red in the Tcl Console after you attempt to run it. Go back to your script and correct the errors that prevented the script from running.
Running Tcl Scripts When Launching the Radiant Software

This topic describes how launch the Radiant software and automatically run Tcl scripts using a command line shell or the stand-alone Tcl console. Your Tcl script can be standard Tcl commands as well as the Radiant software-specific Tcl commands.

Refer to “Creating and Running Custom Tcl Scripts” on page 153 for more information on creating custom Tcl scripts.

To launch the Radiant software and run a Tcl script from a command line shell or the stand-alone Tcl console:

Enter the following command:

On Windows:
```
pnmain.exe -t <tcl_path_file>
```

On Linux:
```
radiant -t <tcl_path_file>
```

Sample Radiant software Tcl Script  The following Radiant software Tcl script shows a very simple script, running in Windows, that opens a project and runs the design flow through the MAP process. Use this simple example as a general guideline.

Figure 93: Simple Radiant Software Script
```
prj_open C:/test/iobasic_radiant/io1.rdf
prj_run_map
```

The above example is saved in Windows as the file mytcl.tcl in the directory C:/test. By running the following command from either a DOS shell or the Tcl console in Windows, the Radiant software GUI starts, the project io1.rdf opens, and the MAP process automatically runs.
```
pnmain.exe -t c:/test/mytcl.tcl
```
Radiant Software Tool Tcl Command Syntax

This part of the Tcl Command Reference Guide introduces the syntax of each of the Radiant software tools and provides you with examples to help you construct your own commands and scripts.

The Radiant software tries to make it easy to develop TCL scripts by mirroring the correct command syntax in the Tcl Console based on the actions performed by you in the GUI. This process works well for most designs, but there are times when a greater degree of control is required. More control over the build process is made available through additional command line switches. The additional switches may not be invoked by actions taken by you when using the GUI. This section provides additional information about all of the Tcl commands implemented in the Radiant software.

The Tcl Commands are broken into major categories. The major categories are:

- Radiant Software Tcl Console Commands
- Radiant Software Timing Constraints Tcl Commands
- Radiant Software Physical Constraints Tcl Commands
- Radiant Software Project Tcl Commands
- Reveal Inserter Tcl Commands
- Reveal Analyzer Tcl Commands
- Power Calculator Tcl Commands
- Programmer Tcl Commands
- Engineering Change Order Tcl Commands

Radiant Software Tcl Console Commands

The Radiant software Tcl Console provides a small number of commands that allow you to perform some basic actions upon the Tcl Console Pane. The Radiant software Tcl Console commands differ from the other Tcl commands.

See Also

- “Running the Tcl Console” on page 151
- “Radiant Software Tool Tcl Command Syntax” on page 157
- “Creating and Running Custom Tcl Scripts” on page 153
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)
provided in the Radiant software. This dtc program’s general Tcl Console commands do not use the dtc_ prefix in the command syntax as is the convention with other tools in the Radiant software.

**Note**

TCL Command Log is always listed after the project is closed. You can find it in the Reports section under Misc Report > TCL Command Log.

The following table provides a listing of all valid Radiant software Tcl Console-related commands.

**Table 18: Radiant Software Tcl Console Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
</table>
| clear   | N/A       | The `clear` command erases anything present in the Tcl Console pane, and prints the current prompt character in the upper left corner of the Tcl Console pane without erasing the command history.  
**It’s only used in GUI Tcl console and not supported in stand-alone Tcl console.** |
| history | N/A       | The `history` command lists the command history in the Tcl Console that you executed in the current session.  
Every command entered into the Tcl Console, either by the GUI, or by direct entry in the Tcl Console, is recorded so that it can be recalled at any time.  
The command history list is cleared when a project is opened or when the Tcl Console `reset` command is executed. |
| reset   | N/A       | The `reset` command clears anything present in the Tcl Console pane, and erases all entries in the command line history.  
**It’s only used in GUI Tcl console and not supported in stand-alone Tcl console.** |
Table 18: Radiant Software Tcl Console Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
</table>
| save_script | <filename.ext> | Saves the contents of the command line history memory buffer into the script file specified. The script is, by default, stored into the current working directory. File paths using forward slashes used with an identifier are valid if using an absolute file path to an existing script folder.  
**It’s only used in GUI Tcl console and not supported in stand-alone Tcl console.** |
| set_prompt  | <new_character> | The default prompt character in the Tcl Console is the “greater than” symbol or angle bracket (i.e., >). You can change this prompt character to some other special character such as a dollar sign ($) or number symbol (#) if you prefer.  
**It’s only used in GUI Tcl console and not supported in stand-alone Tcl console.** |

**Radiant Software Tcl Console Command Examples** This section illustrates and describes a few samples of Radiant Tcl Console commands.

**Example 1** To save a script, you simply use the `save_script` command in the Tcl Console window with a name or file path/name argument. In the first example command line, the file path is absolute, that is, it includes the entire path. Here you are saving "myscript.tcl" to the existing current working directory. The second example creates the same "myscript.tcl" file in the current working directory.

```
save_script C:/lscc/radiant/myproject/scripts/myscript.tcl
```

```
save_script myscript.tcl
```

See “Creating and Running Custom Tcl Scripts” on page 153 for details on how to save and run scripts in the Radiant software.

**Example 2** The following `set_prompt` command reassigns the prompt symbol on the command line as a dollar sign ($). The default is an angle bracket or “greater than” sign (>).

```
set_prompt $
```

**Example 3** The following `history` command will print all of the command history that was recorded in the current Tcl Console session.

```
history
```
## Radiant Software Timing Constraints Tcl Commands

The following table provides a listing of all valid Radiant software Timing Constraints Tcl commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_clock</td>
<td>create_clock -period &lt;period_value&gt; [-name &lt;clock_name&gt;] [-waveform &lt;edge_list&gt;] [＜port_list</td>
<td>pin_list</td>
</tr>
<tr>
<td>create_generated_clock</td>
<td>create_generated_clock [-name &lt;clock_name&gt;] -source &lt;master_pin&gt;[-edges &lt;edge_list&gt;] [-divide_by &lt;factor&gt;] [-multiply_by &lt;factor&gt;] [-duty_cycle &lt;percent&gt;] [-invert][＜pin_list</td>
<td>net_list</td>
</tr>
<tr>
<td>ldc_define_attribute</td>
<td>ldc_define_attribute -attr &lt;attr_type&gt; -value &lt;attr_value&gt; -object_type &lt;object_type&gt; -object &lt;object&gt; [-disable] [-comment &lt;comment&gt;]</td>
<td>Set LSE synthesis attributes for given objects.</td>
</tr>
<tr>
<td>set_clock_groups</td>
<td>set_clock_groups -group &lt;clock_list&gt; &lt;-logically_exclusive</td>
<td>-physically_exclusive</td>
</tr>
<tr>
<td>set_clock_latency</td>
<td>set_clock_latency [-rise] [-fall] [-early</td>
<td>-late] &lt;source&gt; &lt;latency&gt; &lt;object_list&gt;</td>
</tr>
</tbody>
</table>
### Table 19: Radiant Software Timing Constraints Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_false_path</td>
<td>set_false_path [-from &lt;port_list</td>
<td>pin_list</td>
</tr>
<tr>
<td>set_input_delay</td>
<td>set_input_delay -clock &lt;clock_name&gt; [-clock_fall] [-max] [-min] [-add_delay] &lt;delay_value&gt; &lt;port_list&gt;</td>
<td>Set input delay on ports</td>
</tr>
<tr>
<td>set_load</td>
<td>set_load &lt;capacitance&gt; &lt;objects&gt;</td>
<td>Commands to set capacitance on ports</td>
</tr>
<tr>
<td>set_max_delay</td>
<td>set_max_delay [-from &lt;port_list</td>
<td>pin_list</td>
</tr>
<tr>
<td>set_min_delay</td>
<td>set_min_delay [-from &lt;port_list</td>
<td>pin_list</td>
</tr>
</tbody>
</table>
The following table provides a listing of all valid Radiant software Physical Constraints Tcl commands.

### Table 19: Radiant Software Timing Constraints Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
</table>
| set_multicycle_path  | set_multicycle_path [-from <port_list|pin_list|instance_list|net_list|clock_list>]<to <port_list|pin_list|instance_list|net_list|clock_list>]
|                      | [-through <port_list|pin_list|instance_list|net_list|clock_list>] [-rise_from <clock_list>] [-rise_to <clock_list>]<fall_from <clock_list> [-fall_to <clock_list>]
|                      | [-setup|-hold] [-start|-end] <path_multiplier>                             | Define multicycle path          |

### Table 20: Radiant Software Physical Constraints Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldc_create_group</td>
<td>ldc_create_group -name &lt;group_name&gt; [-bbox {height width}] &lt;objects&gt;</td>
<td>Defines a single identifier that refers to a group of objects</td>
</tr>
<tr>
<td>ldc_create_region</td>
<td>ldc_create_region -name &lt;region_name&gt; -site &lt;site&gt; -width &lt;width&gt; -height &lt;height&gt;</td>
<td>Define a rectangular area</td>
</tr>
<tr>
<td>ldc_create_vref</td>
<td>ldc_create_vref -name &lt;vref_name&gt; -site &lt;site_name&gt;</td>
<td>Define a voltage reference</td>
</tr>
<tr>
<td>ldc_prohibit</td>
<td>ldc_prohibit -site &lt;site&gt; -region &lt;region&gt;</td>
<td>Prohibits the use of a site or all sites inside a region</td>
</tr>
<tr>
<td>ldc_set_attribute</td>
<td>ldc_set_attribute &lt;key-value list&gt; [objects]</td>
<td>Set object attributes</td>
</tr>
</tbody>
</table>
Radiant Software Physical Constraints Tcl Commands Examples

Example 1

The following `ldc_create_group` command creates a sample group with 3 instances, and places all instances within the group to a 2x2 bbox.

```tcl
ldc_create_group -name sample_group -bbox {2 2} [get_cells {i16_1_lut i18_2_lut i25_3_lut }]
```

Example 2

The following `ldc_set_location` command places the port clk to pin E7.

```tcl
ldc_set_location -site {E7} [get_ports clk]
```

Example 3

The following `ldc_set_port` command sets IO_TYPE, DRIVE, SLEWRATE attributes of the port rst.

```tcl
ldc_set_port -iobuf {IO_TYPE=LVCMOS33 DRIVE=8 SLEWRATE=FAST} [get_ports rst]
```
Radiant Software Project Tcl Commands

The Radiant software Project Tcl Commands allow you to control the contents and settings applied to the tools, and source associated with your design. Projects can be opened, closed, and configured to a consistent state using the commands described in this section.

Radiant Software Project Tcl Command Descriptions

The following table provides a listing of all valid Radiant software project-related Tcl command options and describes option functionality.

Table 21: Radiant Software Project Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prj_create</td>
<td>prj_create -name &lt;project name&gt; [-dev &lt;device name&gt;] [-performance &lt;performance grade&gt;] [-impl &lt;initial implementation name&gt;] [impl_dir &lt;initial implementation directory&gt;] [-synthesis &lt;synthesis tool name&gt;]</td>
<td>Creates a new project inside the current working directory. The new command can only be used when no other project is currently open. The -name &lt;project name&gt; argument specifies the name of the project. This creates a &lt;project name&gt;.rdf file in the current working directory. The -impl &lt;initial implementation name&gt; argument specifies the active implementation when the project is created. If this left unspecified a default implementation called &quot;Implementation0&quot; is created. The -dev &lt;device name&gt; argument specifies the FPGA family, density, footprint, performance grade, and temperature grade to generate designs for. Use the Lattice OPN (Ordering Part Number) for the &lt;device name&gt; argument. The -performance &lt;performance grade&gt; argument specifies the device performance grade explicitly. For iCE40UP device, performance grade can't be inferred from the device part name such as iCE40UP3K-UWG30ITR. If no performance grade specified, default performance value is used. The -impl_dir &lt;initial implementation directory&gt; argument defines the directory where temporary files are stored. If this is not specified the current working directory is used.</td>
</tr>
<tr>
<td>prj_close</td>
<td>prj_close</td>
<td>Exits the current project. Any unsaved changes are discarded.</td>
</tr>
</tbody>
</table>
### Table 21: Radiant Software Project Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prj_open</td>
<td>prj_open &lt;projectfile.rdf&gt;</td>
<td>Opens the specified project in the software environment.</td>
</tr>
<tr>
<td>prj_save</td>
<td>prj_save [projectfile.rdf]</td>
<td>Updates the project with all changes made during the current session and the project file is saved.</td>
</tr>
<tr>
<td>prj_saveas</td>
<td>prj_saveas -name &lt;new project name&gt; -dir &lt;new project directory&gt; [-copy_gen]</td>
<td>Save the current project as a new project with specified name and directory.</td>
</tr>
<tr>
<td>prj_set_opt</td>
<td>prj_set_opt</td>
<td>List, set or remove a project option.</td>
</tr>
<tr>
<td>prj_archive</td>
<td>prj_archive [-includeAll] &lt;archive_file&gt;</td>
<td>Archive the current project.</td>
</tr>
</tbody>
</table>

---

Lattice Radiant Software 2.0 User Guide 165
### Table 21: Radiant Software Project Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prj_add_source</td>
<td>prj_add_source [-impl &lt;implement name&gt;] [-simulate_only] [-synthesis_only]</td>
<td>Adds a VHDL source file to the specified or active implementation. The syntax used for the Add function depends upon the source file's implementation language.</td>
</tr>
<tr>
<td></td>
<td>[-include &lt;path list for Verilog include search path&gt;]</td>
<td>[-work &lt;VHDL lib name&gt;]: Assigns the source code to the specified library name space.</td>
</tr>
<tr>
<td></td>
<td>[-impl &lt;implementation name&gt;] [-opt &lt;name=value&gt;] ... [-exclude] &lt;src file&gt;...</td>
<td>[-impl &lt;implementation name&gt;]: This switch is used to add a source file to a Radiant software implementation. If this switch is not specified the source file is added to the active implementation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[-opt name=value]: The -opt argument allows you to set a custom, user-defined option. See Example 7 for guidelines and usage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;src file&gt;...: One or more VHDL source files to add to the specified implementation.</td>
</tr>
<tr>
<td>prj_enable_source</td>
<td>prj_enable_source [-impl &lt;implement name&gt;] &lt;src file&gt; ...</td>
<td>Enables the excluded design sources from the current project, that is, it will activate a source file for synthesis, to be used as a constraint, or for Reveal debugging.</td>
</tr>
<tr>
<td>prj_disable_source</td>
<td>prj_disable_source [-impl &lt;implement name&gt;] &lt;src file&gt; ...</td>
<td>Disables the excluded design sources from the current project, that is, it will activate a source file for synthesis, to be used as a constraint or for Reveal debugging.</td>
</tr>
<tr>
<td>prj_remove_source</td>
<td>prj_remove_source [-impl &lt;implement name&gt;] -all &lt;src file&gt; ...</td>
<td>Deletes the specified source files from the specified implementation. If an implementation is not listed explicitly the source files are removed from the active implementation. The source files are not removed from the file system, they are only removed from consideration in the specified implementation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:Remove all the design sources in project</td>
</tr>
<tr>
<td></td>
<td></td>
<td>prj_remove_source [-impl &lt;implement name&gt;] &lt;src file&gt;</td>
</tr>
</tbody>
</table>
Table 21: Radiant Software Project Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prj_set_source_opt</td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>List, set or remove a source option.</td>
</tr>
<tr>
<td></td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>: List all the options in the specified source</td>
</tr>
<tr>
<td></td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
</tr>
<tr>
<td></td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>: List or set the source's option value</td>
</tr>
<tr>
<td></td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
</tr>
<tr>
<td></td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>: Append a value to the specified option value</td>
</tr>
<tr>
<td></td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
</tr>
<tr>
<td></td>
<td>prj_set_source_opt -src &lt;source name&gt; [-impl &lt;implement name&gt;]</td>
<td>: Remove the options of the source</td>
</tr>
<tr>
<td>prj_syn_sim_source</td>
<td>prj_syn_sim_source [-impl &lt;implement name&gt;] -src &lt;source name&gt; [SimulateOnly</td>
<td>SynthesisOnly</td>
</tr>
<tr>
<td>prj_create_impl</td>
<td>prj_create_impl &lt;new impl name&gt; [-dir &lt;implementation directory&gt;] [-strategy &lt;default strategy name&gt;] [-synthesis &lt;synthesis tool name&gt;]</td>
<td>Create a new implementation in the current project with ‘&lt;new impl name&gt;’. The new implementation will use the current active implementation’s strategy as the default strategy if no valid strategy is set.</td>
</tr>
<tr>
<td>prj_remove_impl</td>
<td>prj_remove_impl &lt;implement name&gt;</td>
<td>Delete the specified implementation in the current project with ‘&lt; impl name&gt;’.</td>
</tr>
<tr>
<td>Command</td>
<td>Function (Argument)</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>prj_set_impl_opt</td>
<td>prj_set_impl_opt [-impl &lt;implement name&gt;]</td>
<td>Allows you to add, list, or remove implementation options with the name &lt;implement name&gt; in the specified or active implementation of the current project. If the -rem option is used, the following option names appearing after it will be removed. If no argument is used (i.e., &quot;prj_impl option&quot;), the default is to list all implementation options. If only the &lt;option name&gt; argument is used (i.e., &quot;prj_impl option &lt;option name&gt;&quot;), then the value of that option in the project will be returned. The command will set the option value to the option specified by &lt;option name&gt;. If the &lt;option value&gt; is empty then the option will be removed and ignored (e.g., prj_impl option -rem). The -run_flow argument allows you to switch from the normal mode to an &quot;initial&quot; incremental flow mode and &quot;incremental&quot; which is the mode you should be in after an initial design run during the incremental design flow. With no value parameters specified, -run_flow will return the current mode setting.</td>
</tr>
<tr>
<td></td>
<td>prj_set_impl_opt [-impl &lt;implement name&gt;] &lt;option name&gt; [option value list]</td>
<td>List or set the implementation's option value</td>
</tr>
<tr>
<td></td>
<td>prj_set_impl_opt [-impl &lt;implement name&gt;] -append &lt;option name&gt; &lt;option value&gt;</td>
<td>Append a value to the specified option value</td>
</tr>
<tr>
<td></td>
<td>prj_set_impl_opt [-impl &lt;implement name&gt;] -rem &lt;option name&gt;...</td>
<td>Remove the the options in the implementation</td>
</tr>
<tr>
<td>prj_set_prescript</td>
<td>prj_set_prescript [-impl &lt;implement name&gt;] &lt;milestone name&gt; &lt;script_file&gt;</td>
<td>List or set user Tcl script before running milestone.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>milestone name can be 'syn', 'map', 'par', 'export'</td>
</tr>
<tr>
<td>prj_set_postscript</td>
<td>prj_set_postscript [-impl &lt;implement name&gt;] &lt;milestone name&gt; &lt;script_file&gt;</td>
<td>List or set user Tcl script after running milestone.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>milestone name can be 'syn', 'map', 'par', 'export'</td>
</tr>
<tr>
<td>prj_activate_impl</td>
<td>prj_activate_impl &lt;implement name&gt;</td>
<td>Activates the implementation with the name &lt;implement name&gt;.</td>
</tr>
<tr>
<td>prj_clean_impl</td>
<td>prj_clean_impl [-impl &lt;implement name&gt;]</td>
<td>Clean up the implementation result files in the current project.</td>
</tr>
<tr>
<td>prj_clone_impl</td>
<td>prj_clone_impl &lt;new impl name&gt; [-dir &lt;new impl directory&gt;] [-copyRef] [-impl &lt;original impl name&gt;]</td>
<td>Clone an existing implementation.</td>
</tr>
<tr>
<td>prj_run_synthesis</td>
<td>prj_run_synthesis</td>
<td>Run synthesis process.</td>
</tr>
</tbody>
</table>
Radiant Software Project Tcl Command Examples  This section illustrates and describes a few samples of Radiant software Project Tcl commands.

Example 1  To create a new project, your command may appear something like the following which shows the creation of a ThunderPlus device.

```
prj_create -name "m" -impl "m" -dev iCE40UP3K-UWG30ITR
```

Example 2  To save a project and give it a certain name (save as), use the project save command as shown below:

```
prj_save "my_project"
```

To simply save the current project just use the save function with no values:

```
prj_save
```

Example 3  To open an existing project, the command syntax would appear with the absolute file path on your system as shown in the following example:

```
prj_open "C:/projects/radiant/adder/my_project.rdf"
```

Example 4  To add a source file, in this case a source LDC file, use the prj_src add command as shown below and specify the complete file path:

```
prj_add_source "C:/my_project/radiant/counter/counter.ldc"
```
Example 5  The following examples below shows the prj_run command being used:

prj_run_par

In this final example, synthesis is run.

prj_run_synthesis

Example 6  To copy another project strategy that is already established in another Radiant software project from your console, use the prj_copy_strategy copy command as shown below and specify the new strategy name and the strategy file name.

prj_copy_strategy -from source_strategy -name new_strategy -file strategy.stg

Example 7  The prj_add_source command allows you to set a custom, user-defined option. This -opt argument value, however, cannot conflict with existing options already in the system, that is, its identifier must differ from system commands such as "include" and "lib" for example. In addition, a user-defined option may not affect the internal flow but can be queried for any usage in a user's script to arrange their design and sources. All user-defined options can be written to the Radiant software project RDF file.

In the example below, the -opt argument is used as a qualifier to make a distinction between to .rvl file test cases.

prj_add_source test1.rvl -opt "debug_case=golden_case"
prj_add_source test2.rvl -opt "debug_case=bad_case"

Example 8  After you modify your strategy settings in the Radiant software interface the values are saved to the current setting via a Tcl command. For example, a command similar to the following will be called if Synplify frequency and area options are changed.

prj_set_strategy_value -strategy strategy1 SYN_Frequency=300 SYN_Area=False

Simulation Libraries Compilation Tcl Commands
This section provides Simulation Libraries Compilation extended Tcl command syntax and usage examples.
Simulation Libraries Compilation Tcl Command Descriptions

The following table provides a listing of all valid Simulation Libraries Compilation Tcl Command arguments and describes their usage.

**Note**

Running cmpl_libs may take a long time and may cause the Radiant software to hang.

- It is recommended to run cmpl_libs using the Radiant TCL Console (Start Menu > Lattice Radiant Software > Accessories > TCL Console).

or,

- Run cmpl_libs.tcl using the command line console. Refer to “Running cmpl_libs.tcl from the Command Line” on page 107.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpl_libs</td>
<td>-sim_path &lt;sim_path&gt; [-sim_vendor {mentor&lt;default&gt;}] [-device {ice40up</td>
<td>LIFCL</td>
</tr>
</tbody>
</table>
Simulation Libraries Compilation Tcl Command Examples  This section illustrates and describes a few examples of Simulation Libraries Compilation Tcl command.

Example 1  The following command will compile all the Lattice FPGA libraries for both Verilog and VHDL simulation, and place them under the folder specified by -target_path. The path to Modelsim is specified by -sim_path.

```
cmpl_libs -sim_path C:/questasim64_10.4e/win64 -target_path c:/mti_libs
```

Reveal Inserter Tcl Commands

This section provides Reveal Inserter extended Tcl command syntax, command options, and usage examples.

Reveal Inserter Tcl Command Descriptions  The following table provides a listing of all valid Reveal Inserter Tcl command options and describes option functionality.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rvl_new_project</td>
<td>rvl_new_project &lt;rvl file&gt;</td>
<td>Create a new reveal inserter project.</td>
</tr>
<tr>
<td>rvl_open_project</td>
<td>rvl_open_project &lt;rvl file&gt;</td>
<td>Open a reveal inserter project file.</td>
</tr>
<tr>
<td>rvl_save_project</td>
<td>rvl_save_project &lt;rvl file&gt;</td>
<td>Save the current reveal inserter project.</td>
</tr>
<tr>
<td>rvl_close_project</td>
<td>rvl_close_project</td>
<td>Close the current reveal inserter project.</td>
</tr>
<tr>
<td>rvl_run_project</td>
<td>rvl_run_project [-save] [-saveAs &lt;file&gt;] [-overwrite] [-drc] [-insert_core &lt;core_name&gt;]</td>
<td>Run inserting debug core task or DRC checking on the current reveal inserter project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-save: Save the project before run command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-saveAs: Save as a different file before run command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-overwrite: Overwrite the existing file if the saved as to file exists already</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-drc: Run DRC checking only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-insert_core: Specify the core to be inserted. All cores will be inserted if none is specified</td>
</tr>
<tr>
<td>rvl_add_core</td>
<td>rvl_add_core &lt;core name&gt;</td>
<td>Add a new core in current project.</td>
</tr>
<tr>
<td>rvl_del_core</td>
<td>rvl_del_core &lt;core name&gt;</td>
<td>Remove an existing core from current project.</td>
</tr>
</tbody>
</table>
Table 23: Reveal Inserter Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rvl_rename_core</td>
<td>rvl_rename_core &lt;core name&gt;</td>
<td>Rename an existing core from current project.</td>
</tr>
<tr>
<td></td>
<td>&lt;new core name&gt;</td>
<td></td>
</tr>
<tr>
<td>rvl_set_core</td>
<td>rvl_set_core [core name]</td>
<td>List the default core or select a core as the default core in current project.</td>
</tr>
<tr>
<td>rvl_list_core</td>
<td>rvl_list_core</td>
<td>List all cores in current project.</td>
</tr>
<tr>
<td>rvl_add_serdes</td>
<td>rvl_add_serdes</td>
<td>Add the Serdes core into current project.</td>
</tr>
<tr>
<td>rvl_del_serdes</td>
<td>rvl_del_serdes</td>
<td>Remove the Serdes core from current project.</td>
</tr>
<tr>
<td>rvl_set_serdes</td>
<td>rvl_set_serdes [clk=&lt;clock name&gt;] [rst=&lt;reset signal, default value is VLO&gt;]</td>
<td>List or set options of Serdes core.</td>
</tr>
<tr>
<td>rvl_add_controller</td>
<td>rvl_add_controller</td>
<td>Add the Controller Core into current project.</td>
</tr>
<tr>
<td>rvl_del_controller</td>
<td>rvl_del_controller</td>
<td>Remove the Controller Core from current project.</td>
</tr>
<tr>
<td>rvl_set_controller</td>
<td>rvl_set_controller {[-item LED</td>
<td>Switch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can set opt_list with the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Insert=[on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Width=[1..32] for LED and Switch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ AddrWidth=[4..16] for Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ DataWidth=[4..32] for Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sig_list with the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ SWn=signal where n=1 to Width for Switch.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ LEDn=signal where n=1 to Width for LED.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Clock=clk_signal for Register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Enable=en_signal for Register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Wr_Rdn=wr_rdn_signal for Register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Address=addr_bus for Register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ WData=wdata_bus for Register.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ RData=rdata_bus for Register.</td>
</tr>
<tr>
<td>rvl_add_trace</td>
<td>rvl_add_trace [-core &lt;core name&gt;] [-insert_at &lt;position&gt;] &lt;signals list&gt;</td>
<td>Add trace signals in a debug core in current project. You can specify an existing trace signal/bus name or a position number in a trace bus as the inserting position.</td>
</tr>
<tr>
<td>rvl_del_trace</td>
<td>rvl_del_trace [-core &lt;core name&gt;] &lt;signals list&gt;</td>
<td>Delete trace signals in a debug core in current project.</td>
</tr>
</tbody>
</table>
Table 23: Reveal Inserter Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rvl_rename_trace</td>
<td>rvl_rename_trace [-core &lt;core name&gt;] -bus &lt;bus name&gt; &lt;new bus name&gt;</td>
<td>Change the name of a trace bus in a debug core in current project.</td>
</tr>
<tr>
<td>rvl_list_trace</td>
<td>rvl_list_trace [-core &lt;core name&gt;]</td>
<td>List all trace signals in a debug core in current project.</td>
</tr>
<tr>
<td>rvl_move_trace</td>
<td>rvl_move_trace [-core &lt;core name&gt;] [-move_to &lt;position&gt;] &lt;signals list&gt;</td>
<td>Move and rearrange the order of trace signals in a debug core in current project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can specify an existing trace signal/bus name or a postion number in a trace bus as the new position.</td>
</tr>
<tr>
<td>rvl_group_trace</td>
<td>rvl_group_trace [-core &lt;core name&gt;] -bus &lt;bus name&gt; &lt;signals list&gt;</td>
<td>Group specified trace signals in a debug core in current project into a bus.</td>
</tr>
<tr>
<td>rvl_ungroup_trace</td>
<td>rvl_ungroup_trace [-core &lt;core name&gt;] &lt;bus name&gt;</td>
<td>Ungroup trace signals in a trace bus in a debug core in current project.</td>
</tr>
<tr>
<td>rvl_set_traceoptn</td>
<td>rvl_set_traceoptn [-core &lt;core name&gt;] [option=value]</td>
<td>List or set trace options of a debug core in current project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can set the following option:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SampleClk = [signal name].</td>
</tr>
<tr>
<td>rvl_set_trigoptn</td>
<td>rvl_set_trigoptn [-core &lt;core name&gt;] [option=value]</td>
<td>List or set trigger options of a debug core in current project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can set the following option:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DefaultRadix = [bin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EventCounter = [on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CounterValue = [2,4,8,16,...,65536] (depend on FinalCounter is on)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TriggerOut = [on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OutNetType = [IO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OutNetName = [net name] (depend on TriggerOut is on)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OutNetPri = [Active_Low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OutNetMPW = [pulse number] (depend on TriggerOut is on).</td>
</tr>
<tr>
<td>rvl_list_tu</td>
<td>rvl_list_tu [-core &lt;core name&gt;]</td>
<td>List all trigger units in a debug core in current project.</td>
</tr>
</tbody>
</table>
Table 23: Reveal Inserter Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rvl_add_tu</td>
<td>rvl_add_tu [-core &lt;core name&gt;] [-radix &lt;bin</td>
<td>oct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TU definition format: &quot;{signal list} Operator Value&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operator must be &quot;:=&quot;&quot;,&quot;=&quot;&quot;,&quot;&gt;&quot;,&quot;&gt;=&quot;,&quot;&lt;&quot;,&quot;&lt;=&quot;,&quot;.RE.&quot;(rising edge), &quot;.FE.&quot;(falling edge) and &quot;.SC.&quot;(serial compare).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A default trigger unit name will be created if it's omitted in command.</td>
</tr>
<tr>
<td>rvl_del_tu</td>
<td>rvl_del_tu [-core &lt;core name&gt;] &lt;TU name&gt;</td>
<td>Remove an existing core from current project.</td>
</tr>
<tr>
<td>rvl_rename_tu</td>
<td>rvl_rename_tu [-core &lt;core name&gt;] &lt;old name&gt; &lt;new name&gt;</td>
<td>Rename an existing core in current project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TU definition format: &quot;{signal list} Operator Value&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operator must be &quot;:=&quot;,&quot;=&quot;&quot;,&quot;&gt;&quot;,&quot;&gt;=&quot;,&quot;&lt;&quot;,&quot;&lt;=&quot;,&quot;.RE.&quot;(rising edge), &quot;.FE.&quot;(falling edge) and &quot;.SC.&quot;(serial compare).</td>
</tr>
<tr>
<td>rvl_list_te</td>
<td>rvl_list_te [-core &lt;core name&gt;]</td>
<td>List all trigger expressions in a debug core in current project.</td>
</tr>
<tr>
<td>rvl_add_te</td>
<td>rvl_add_te [-core &lt;core name&gt;] [-ram &lt;EBR</td>
<td>Slice&gt;] [-name &lt;new TE name&gt;] [-expression &lt;expression string&gt;] [-max_seq_depth &lt;max depth&gt;] [-max_event_count &lt;max event count&gt;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A default trigger expression name will be created if it's omitted in command.</td>
</tr>
<tr>
<td>rvl_del_te</td>
<td>rvl_del_te [-core &lt;core name&gt;] &lt;TE name&gt;</td>
<td>Delete an existing trigger expression in a debug core in current project.</td>
</tr>
<tr>
<td>rvl_rename_te</td>
<td>rvl_rename_te [-core &lt;core name&gt;] &lt;old name&gt; &lt;new name&gt;</td>
<td>Rename an existing trigger expression in a debug core in current project.</td>
</tr>
<tr>
<td>rvl_set_te</td>
<td>rvl_set_te [-core &lt;core name&gt;] [-ram &lt;EBR</td>
<td>Slice&gt;] [-expression &lt;expression string&gt;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[-max_seq_depth &lt;max depth&gt;] [-max_event_count &lt;max event count&gt;] &lt;TE name&gt;</td>
</tr>
</tbody>
</table>
Reveal Inserter Tcl Command Examples  This section illustrates and describes a few samples of Reveal Inserter Tcl commands.

Example 1  To create a new Reveal Inserter project with the .rvl file extension in your project directory, use the rvl_project command as shown below using the new option.

```
rvl_new_project my_project.rvl
```

Example 2  The following example shows how to set up TU parameters for Reveal Inserter:

```
rvl_set_tu -name TU -add_sig {count[7:0]} -op == -val C3 -radix Hex
```

Reveal Analyzer Tcl Commands

This section provides Reveal Analyzer extended Tcl command syntax, command options, and usage examples.

Reveal Analyzer Tcl Command Descriptions  The following table provides a listing of all valid Reveal Analyzer Tcl command options and describes option functionality.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rva_new_project</td>
<td>rva_new_project &lt;file&gt;</td>
<td>Create a new Reveal Analyzer project.</td>
</tr>
<tr>
<td>rva_open_project</td>
<td>rva_open_project &lt;file&gt;</td>
<td>Open a Reveal Analyzer project file.</td>
</tr>
<tr>
<td>rva_save_project</td>
<td>rva_save_project &lt;file&gt;</td>
<td>Save the current Reveal Analyzer project.</td>
</tr>
<tr>
<td>rva_close_project</td>
<td>rva_close_project &lt;file&gt;</td>
<td>Close the current Reveal Analyzer project.</td>
</tr>
<tr>
<td>rva_export_project</td>
<td>rva_export_project -vcd &lt;file name&gt;</td>
<td>Export VCD file. Optional to include a title in the VCD file. By default the title will be &quot;&lt;unknown&gt;&quot;.</td>
</tr>
<tr>
<td></td>
<td>[-module &lt;title&gt;]</td>
<td></td>
</tr>
<tr>
<td>rva_export_project</td>
<td>rva_export_project -txt &lt;file name&gt;</td>
<td>Export TEXT file. Optional to export selected signal list only. By default all signals are exported.</td>
</tr>
<tr>
<td></td>
<td>[-siglist &lt;signal list&gt;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 24: Reveal Analyzer Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rva_set_project</td>
<td>rva_set_project [-frequency &lt;val&gt;]</td>
<td>No arguments specified will return options.</td>
</tr>
<tr>
<td></td>
<td>[-period &lt;val&gt;] [-tckdelay &lt;val&gt;]</td>
<td>-frequency: sets the frequency value for sample clock in MHz</td>
</tr>
<tr>
<td></td>
<td>[-cabletype &lt;val&gt;] [-cableport &lt;val&gt;]</td>
<td>-period: sets a period value for sample clock in ns or ps</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-tckdelay: sets a TCK clock pin pulse width delay value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-cabletype: sets the type of cable. Values are LATTICE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-cableport: sets the port number as integer &gt;= 0.</td>
</tr>
<tr>
<td>rva_run</td>
<td>rva_run</td>
<td>Runs until trigger condition to capture data.</td>
</tr>
<tr>
<td>rva_stop</td>
<td>rva_stop</td>
<td>Stops without capturing data.</td>
</tr>
<tr>
<td>rva_manualtrig</td>
<td>rva_manualtrig</td>
<td>Manual Trigger to capture data.</td>
</tr>
<tr>
<td>rva_get_trace</td>
<td>rva_get_trace</td>
<td>Lists all trace signals in a core.</td>
</tr>
<tr>
<td>rva_set_core</td>
<td>rva_set_core [-name &lt;name&gt;] [-run &lt;on</td>
<td>off&gt;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-name: Select core. Needed for other actions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-run: Turns run option on/off for core.</td>
</tr>
<tr>
<td>rva_set_tu</td>
<td>rva_set_tu [-name &lt;name&gt;] [-operator {==</td>
<td>!=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-name: Select TU. If no options, return options and value for the selected TU.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-operator: Sets the comparison operator. Operators are equal to (==), not equal to (!=), greater than (&gt;), greater than or equal to (&gt;=), not equal to (&lt;=), less than (&lt;), less than or equal to (&lt;=), &quot;rising edge&quot;, &quot;falling edge&quot;, and serial compare (serial).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-value: Sets TU value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-radix: Sets TU radix. Options are binary (bin), octal (oct), decimal (dec), hexadecimal (hex), or the name of a token set.</td>
</tr>
<tr>
<td>rva_rename_tu</td>
<td>rva_rename_tu &lt;name&gt; &lt;new name&gt;</td>
<td>This function renames TU.</td>
</tr>
<tr>
<td>rva_set_te</td>
<td>rva_set_te [-name &lt;name&gt;] [-expression &lt;expression list&gt;] [-enable &lt;on</td>
<td>off&gt;]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-name: Select TE. If no options, return options and value for the selected TE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-expression: Sets TE expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-enable: Enables/disables TE.</td>
</tr>
</tbody>
</table>
Table 24: Reveal Analyzer Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rva_rename_te</td>
<td>rva_rename_te &lt;name&gt; &lt;new name&gt;</td>
<td>This function renames TE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-teall: Sets AND ALL or OR ALL for all TEs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-finalcounter: Turns final trigger counter on/off</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-finalcountervalue: Sets final trigger counter value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-samples: Sets number of samples to capture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-numtriggers: Sets number of triggers to capture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-position: Sets trigger position to pre-selected or user value.</td>
</tr>
<tr>
<td>rva_add_token</td>
<td>rva_add_token &lt;tokenset name&gt; &lt;name=value&gt;</td>
<td>Add a token with new name and value in a specific token set.</td>
</tr>
<tr>
<td>rva_del_token</td>
<td>rva_del_token &lt;tokenset name&gt; &lt;token name&gt;</td>
<td>Delete a specific token in a specific token set.</td>
</tr>
<tr>
<td>rva_set_token</td>
<td>rva_set_token &lt;tokenset name&gt; &lt;token name&gt; &lt;name new token name&gt; &lt;value new token value&gt;</td>
<td>Select specific token in specific token set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-name: Set token name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-value: Set token value</td>
</tr>
<tr>
<td>rva_add_tokenset</td>
<td>rva_add_tokenset [-tokenset &lt;tokenset name&gt;] [-bits &lt;token bits&gt;] [-token &lt;name=value&gt;]</td>
<td>No arguments, add a token set with default name and bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-tokenset: Set token set name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-bits: Set token set bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-token: Add extra tokens</td>
</tr>
<tr>
<td>rva_del_tokenset</td>
<td>rva_del_tokenset &lt;tokenset name&gt;</td>
<td>Delete the specific token set.</td>
</tr>
<tr>
<td></td>
<td>rva_del_tokenset -all</td>
<td>Delete all token set</td>
</tr>
<tr>
<td>rva_set_tokenset</td>
<td>rva_set_tokenset &lt;tokenset name&gt; &lt;name new token set name&gt; &lt;bits new token bits&gt;</td>
<td>Select specific token set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-name: Rename a token set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-bits: Set number of bits in tokens</td>
</tr>
<tr>
<td>rva_export_tokenset</td>
<td>rva_export_tokenset &lt;file name&gt;</td>
<td>Export all token set to a specific file.</td>
</tr>
<tr>
<td>rva_import_tokenset</td>
<td>rva_import_tokenset &lt;file name&gt;</td>
<td>Import and merge all token set from a specific file.</td>
</tr>
<tr>
<td>rva_open_pcs</td>
<td>rva_open_pcs</td>
<td>Open connection to Lattice device before read/write begins.</td>
</tr>
<tr>
<td>rva_close_pcs</td>
<td>rva_close_pcs</td>
<td>Close connection to Lattice device after read/write finished.</td>
</tr>
</tbody>
</table>
### Table 24: Reveal Analyzer Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rva_read_pcs</td>
<td>rva_read_pcs -byte 1 -addr &lt;address&gt;</td>
<td>Read one byte of data from address in hex.</td>
</tr>
<tr>
<td>rva_write_pcs</td>
<td>rva_write_pcs -byte 1 -addr &lt;address&gt; -data &lt;value&gt;</td>
<td>Write one byte of data to address in hex.</td>
</tr>
<tr>
<td>rva_run_pcs</td>
<td>rva_run_pcs - config_sram -file &lt;tcl file&gt;</td>
<td>Apply changes to SERDES control. -config_sram: Reload all registers in current DCU with values from config SRAM. -file: Run SERDES commands from Tcl file.</td>
</tr>
<tr>
<td>rva_export_pcs</td>
<td>rva_export_pcs &lt;file name&gt;</td>
<td>Export SRV file.</td>
</tr>
<tr>
<td>rva_import_pcs</td>
<td>rva_import_pcs &lt;file name&gt;</td>
<td>Import SRV file.</td>
</tr>
<tr>
<td>rva_open_controller</td>
<td>rva_open_controller</td>
<td>Open Controller connection to Lattice device before read/write begins.</td>
</tr>
<tr>
<td>rva_target_controller</td>
<td>rva_target_controller</td>
<td>Set Controller core as target before read/write begins.</td>
</tr>
<tr>
<td>rva_close_controller</td>
<td>rva_close_controller</td>
<td>Close Controller connection to Lattice device after read/write finished.</td>
</tr>
<tr>
<td>rva_read_controller</td>
<td>rva_read_controller -addr &lt;addr32&gt;</td>
<td>Read data from 32-bit address in hex.</td>
</tr>
<tr>
<td>rva_write_controller</td>
<td>rva_write_controller -addr &lt;addr32&gt; -data &lt;data&gt;</td>
<td>Write data to 32-bit address in hex.</td>
</tr>
<tr>
<td>rva_set_controller</td>
<td>rva_set_controller -option &lt;value&gt;</td>
<td>Set the options for Controller core. -cable_type: Set type of cable as USB or USB2. -cable_port: Set logical port of cable as integer. If no arguments specified, then return list of options and values.</td>
</tr>
</tbody>
</table>
Reveal Analyzer Tcl Command Examples  This section illustrates and describes a few samples of Reveal Analyzer Tcl commands.

**Example 1**  The following command line example shows how to specify a new project that uses a parallel cable port.

```
rva_new_project -rva untitled -rvl "count.rvl" -dev "LFXP2-5E:0x01299043" -port 888 -cable LATTICE
```

**Example 2**  The following example shows how to set up TU parameters for Reveal Analyzer:

```
rva_set_tu -name TU1 -operator == -value 10110100 -radix bin
```

---

Power Calculator Tcl Commands

This section provides Power Calculator extended Tcl command syntax, command options, and usage examples.

**Power Calculator Tcl Command Descriptions**  The following table provides a listing of all valid Power Calculator Tcl command options and describes option functionality.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwc_new_project</td>
<td>pwc_new_project &lt;file&gt;</td>
<td>Create a new project.</td>
</tr>
<tr>
<td>pwc_open_project</td>
<td>pwc_open_project &lt;file&gt;</td>
<td>Open a project file.</td>
</tr>
<tr>
<td>pwc_save_project</td>
<td>pwc_save_project &lt;file&gt;</td>
<td>Save the current project.</td>
</tr>
<tr>
<td>pwc_close_project</td>
<td>pwc_close_project</td>
<td>Close the current project.</td>
</tr>
<tr>
<td>pwc_set_afpervcd</td>
<td>pwc_set_afpervcd &lt;file&gt;</td>
<td>Open vcd file and set frequency and activity factor.</td>
</tr>
<tr>
<td>pwc_set_device</td>
<td>pwc_set_device -family &lt;family name&gt;</td>
<td>Set family.</td>
</tr>
<tr>
<td></td>
<td>pwc_set_device -device &lt;device name&gt;</td>
<td>Set device.</td>
</tr>
<tr>
<td></td>
<td>pwc_set_device -package &lt;package name&gt;</td>
<td>Set package.</td>
</tr>
</tbody>
</table>
### Table 25: Power Calculator Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwc_set_device</code></td>
<td>-speed <code>&lt;speed name&gt;</code></td>
<td>Set speed.</td>
</tr>
<tr>
<td><code>pwc_set_device</code></td>
<td>-operating <code>&lt;operating name&gt;</code></td>
<td>Set operating.</td>
</tr>
<tr>
<td><code>pwc_set_device</code></td>
<td>-part <code>&lt;part name&gt;</code></td>
<td>Set part.</td>
</tr>
<tr>
<td><code>pwc_set_processstype</code></td>
<td><code>&lt;value&gt;</code></td>
<td>Set device power process type.</td>
</tr>
<tr>
<td><code>pwc_set_ambienttemp</code></td>
<td><code>&lt;value&gt;</code></td>
<td>Set ambient temperature value.</td>
</tr>
<tr>
<td><code>pwc_set_thetaja</code></td>
<td><code>&lt;value&gt;</code></td>
<td>Set user defined theta JA.</td>
</tr>
<tr>
<td><code>pwc_set_freq</code></td>
<td><code>&lt;frequency&gt;</code></td>
<td>Set default frequency.</td>
</tr>
<tr>
<td><code>pwc_set_freq</code></td>
<td>-clock <code>&lt;frequency&gt;</code></td>
<td>Set Clock frequency.</td>
</tr>
<tr>
<td><code>pwc_set_freq</code></td>
<td>-timing <code>&lt;option&gt;</code></td>
<td>Set frequency by timing.</td>
</tr>
<tr>
<td><code>pwc_set_freq</code></td>
<td>option: min</td>
<td>pref</td>
</tr>
<tr>
<td><code>pwc_set_af</code></td>
<td><code>&lt;value&gt;</code></td>
<td>Set default activity factor.</td>
</tr>
<tr>
<td><code>pwc_set_estimation</code></td>
<td><code>&lt;value&gt;</code></td>
<td>Sets estimated routing option.</td>
</tr>
<tr>
<td><code>pwc_set_supply</code></td>
<td><code>-type </code>&lt;value&gt;<code>-voltage</code>&lt;value&gt;<code>-dpm</code>&lt;value&gt;`</td>
<td>Set multiplication factor and voltage of named power supply.</td>
</tr>
<tr>
<td><code>pwc_add_ipblock</code></td>
<td><code>-iptype </code>&lt;iptype name&gt;`</td>
<td>Add IP Block row.</td>
</tr>
<tr>
<td><code>pwc_set_ipblock</code></td>
<td><code>-iptype </code>&lt;iptype name&gt;<code>-matchkeys</code>{&lt;key1&gt;<code> </code>&lt;value1&gt;<code>}+ -setkey </code>&lt;key&gt;<code> </code>&lt;value&gt;`: iptypename mapping to PGT section, key mapping to _KEY in pgt session, value is its value</td>
<td>Set IP Block row.</td>
</tr>
<tr>
<td><code>pwc_remove_ipblock</code></td>
<td><code>-iptype </code>&lt;iptype name&gt;<code>-matchkeys</code>{&lt;key1&gt;<code> </code>&lt;value1&gt;`}+</td>
<td>Remove IP Block row.</td>
</tr>
<tr>
<td><code>pwc_gen_report</code></td>
<td><code>&lt;file&gt;</code></td>
<td>Generate text report and write to file.</td>
</tr>
<tr>
<td><code>pwc_gen_htmlreport</code></td>
<td><code>&lt;file&gt;</code></td>
<td>Generate HTML report and write to file.</td>
</tr>
</tbody>
</table>

**Power Calculator Tcl Command Examples** This section illustrates and describes a few samples of Power Calculator Tcl commands.

**Example 1** The follow command below creates a PWC project (.pcf) file named “abc.pcf” from an input UDB file named “abc.UDB”:

```
pwc_new_project abc.pcf -udb abc.udb
```
Example 2  To set the default frequency to, for example, 100 Mhz:

```
pwc_set_freq 100
```

Example 3  The command below saves the current project to a new name:

```
pwc_save_project newname.pcf
```

Example 4  To create an HTML report, you would run a command like the one shown below:

```
pwc_gen_htmlreport c:/abc.html
```

**Programmer Tcl Commands**

This section provides the Programmer extended Tcl command syntax, command options, and usage examples. The below commands are only supported in standalone Programmer currently.

**Programmer Tcl Command Descriptions**  The following table provides a listing of all valid Programmer Tcl command options and describes option functionality.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgr_project</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pgr_project open &lt;project_file&gt;</td>
<td>The open command will open the specified project file in-memory.</td>
</tr>
<tr>
<td></td>
<td>pgr_project save [file_path]</td>
<td>Writes the current project to the specified path. If there is no file path specified then it will overwrite the original file.</td>
</tr>
<tr>
<td></td>
<td>pgr_project close</td>
<td>Closes the current project. If a Programmer GUI is open with the associated project, then the corresponding Programmer GUI will be closed as well.</td>
</tr>
<tr>
<td></td>
<td>pgr_project help</td>
<td>Displays help for the pgr_project command.</td>
</tr>
</tbody>
</table>
Programmer Tcl Command Examples

This section illustrates and describes a few samples of Programmer Tcl commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pgr_program</strong></td>
<td>&lt;no_argument&gt;</td>
<td>When pgr_program is run without arguments it will display the current status of the available settings. Note that specifying a key without a value will display the current value. The following keys can be used to modify those settings. Generally, the pgr_program command and its sub-commands allow you to run the equivalent process commands from the TCL Console window in the Radiant software interface. These commands can override connection options that are set in user defaults.</td>
</tr>
<tr>
<td><strong>pgr_program set -cable</strong></td>
<td>&lt;LATTICE</td>
<td>USB</td>
</tr>
<tr>
<td><strong>pgr_program set -portaddress</strong></td>
<td>&lt;0x0378</td>
<td>0x0278</td>
</tr>
<tr>
<td><strong>pgr_program run</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pgr_program help</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pgr_genfile</strong></td>
<td>&lt;no_argument&gt;</td>
<td>Programmer generate files command (not supported for customer use)</td>
</tr>
<tr>
<td><strong>pgr_genfile set -process</strong></td>
<td>&lt;svf</td>
<td>vme12&gt;</td>
</tr>
<tr>
<td><strong>pgr_genfile set -outfile</strong></td>
<td>&lt;file path&gt;</td>
<td>Sets the output file.</td>
</tr>
<tr>
<td><strong>pgr_genfile run</strong></td>
<td></td>
<td>Generates file based on the current xcf and current settings.</td>
</tr>
<tr>
<td><strong>pgr_genfile help</strong></td>
<td></td>
<td>Displays help for pgr_genfile command.</td>
</tr>
</tbody>
</table>

Table 26: Programmer Tcl Commands
Example 1  The first command below opens a Programmer XCF project file that exists in the system. There can be many programming files associated with one project. In the GUI interface, the boldfaced file in the Radiant software is the active project file.

```
pgr_project open /home/mdm/config_file/myfile.xcf
```

Example 2  The following command sets programming option using a USB2 cable at port address “FTUSB-1, then using pgr_program run to program”.

```
pgr_program set -cable USB2 -portaddress FTUSB-1
```

Example 3  The following command sets the file generation type for JTAG SVF file, then using pgr_genfile run to generates an output file “mygenfile.svf” in a relative path.

```
pgr_genfile set -process svf -outfile ../genfiles/mygenfile.svf
```

### Engineering Change Order Tcl Commands

This section provides Engineering Change Order (ECO) extended Tcl command syntax, command options, and usage examples.

**ECO Tcl Command Descriptions**  The following table provides a listing of all valid ECO Tcl command options and describes option functionality.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function (Argument)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eco_save_design</td>
<td>eco_save_design [-udb &lt;udb_file&gt;]</td>
<td>Saves an existing design or macro.</td>
</tr>
<tr>
<td>eco_config_sysio</td>
<td>eco_config_sysio -comp &lt;comp name&gt; {&lt;key=value&gt;}...</td>
<td>Config sysio setting.</td>
</tr>
<tr>
<td>eco_config_memory</td>
<td>eco_config_memory -mem_id &lt;memory_id&gt; {-init_file &lt;mem_file&gt; - format HEX</td>
<td>BIN</td>
</tr>
</tbody>
</table>

**ECO Tcl Command Examples**  This section illustrates and describes a few samples of ECO Tcl commands.

**Example 1**  The following demonstrates the sysio command:

```
eco_config_sysio -comp {data} {clamp=OFF;diffdrive=NA;diffresistor=OFF;drive=2;glitchfilter=OFF;hysteresis=NA;opendrain=OFF;pullmode=NONE;slewrate=SLOW;termination=OFF;vref=OFF}
```
Example 2  The following demonstrates the memory command:

```
eco_config_mem -mem_id {mem} -init_file {D:/mem/init_hex.mem} -format HEX
```
Chapter 10

Advanced Topics

This chapter explains advanced concepts, features and operational methods for the Radiant software.

Shared Memory Environment

The Radiant software design environment uses a shared memory architecture. Shared memory allows all internal tool views to access the same image of the design at any point in time. Understanding how shared memory is being used can give you insight into managing the environment for optimum performance, especially when your design is large.

There is one shared database that contains the device, design, and constraint information in system memory.

Generating the hierarchy of your design uses an additional database separate from the primary shared memory database.

External tools referenced from within the Radiant software, such as those for synthesis and simulation, use their own memory in addition to what is used by the Radiant software.

Because it is accessing shared memory, the initial tool view launch will take longer than the launch of subsequent views.

Clear Tool Memory

The “Clear Tool Memory command, available from the Tools menu, clears the device, design, and constraint information from system memory. Clearing the tool memory can speed up memory-intensive processes such as place and route. When your design is very large, it is good practice to clear memory prior to running place and route.

If you have open tool views that will be affected by clearing the tool memory, a confirmation dialog box will open to give you the opportunity to cancel the memory clear.
Environment and Tool Options

The Radiant software provides many environment control and tool view display options that enable you to customize your settings. Choose Tools > Options to access these options.

The Options dialog box is organized into functional folders.
Commonly configured items include:

- **General settings** -- allows you to set some common options, including:
  - Startup – enables you to configure the default action at startup and also to control the frequency of checking for software updates.
  - File Associations – allows you to set the programs to be associated with different file types based on the file extensions.
  - Directories – Set directory location for Synthesis and Simulation tools.
  - Network Settings – Apply a proxy server and specify a host and port.

- **Color settings** -- allows you to set colors for such GUI features as fonts and backgrounds for various Radiant software tools. You can also change the Theme color of the Radiant software (Dark or Light) using the Themes drop-down menu.

- **Tools settings** -- allows you to set options for various Radiant software tools, including the Device Constraint Editor, Netlist Analyzer, Timing Constraint Editor, and Source Editor. You can also set the constraint design rule check (DRC) time to real time, prior to saving or when launching a tool.

### Batch Tool Operation

The core tools in the FPGA implementation design flow can all be run in batch mode using command-line tool invocation or scripts. For detailed information, see the *Command Line Reference Guide*, available from the Radiant software Start Page.

### Tcl Scripts

The Radiant Extended Tcl language enables you to create customized scripts for tasks that you perform often in the Radiant software. Automating these operations through Tcl scripts not only saves time, but also provides a uniform approach to design. This is especially useful when you try to find an optimal solution using numerous design implementations.

### Creating Tcl Scripts from Command History

A good first step in Tcl programming is to create a Tcl script by saving some command history and then modifying it as needed. This allows you to get started by using existing command information.

**To create a Tcl command script using command history:**

1. In the Tcl Console window, perform a *reset* command so that your script won’t contain any of the actions that may have already been executed.
reset

2. Open the project and perform the commands that you want to save as a script.

3. Optionally, enter the history command in the Tcl Console window to ensure that the commands you wish to save are in the console’s memory.
   In the Tcl Console window type
   
   ```
   save_script <filename.tcl>
   ```
   
   The `<filename.tcl>` can be any file identifier that has no spaces and contains no special characters except underscores. For example, `myscript.tcl` or `design_flow_1.tcl` are acceptable save_script values, but `my$script` or `my script` are invalid. A file name with an extension of '.ext' will not work.
   
   The `<filename.ext>` entry can be preceded with an absolute or relative path. Use the "/" (i.e. forward slash) symbol to delimit the path elements.
   
   If the path is not specified, the script is saved in the current working directory. The current working directory can be determined by using the TCL `pwd` command.

4. Navigate to your script file and use the text editing tool of your choice to make any necessary changes, such as deleting extraneous lines or invalid arguments.

In most cases, you will need to edit the script you saved and take out any invalid arguments or any commands that cannot be performed in the Radiant software environment due to a conflict or exception. You will likely have to revisit this step later if after running your script, you experience any run errors due to syntax errors or technology exceptions.

### Creating Tcl Scripts from Scratch

Tcl commands can be written directly into a script file. You can use any text editor, such as Notepad or vi, to create a file and type in the Tcl commands.

### Sample Tcl Script

The following Tcl example shows a simple script that opens a project, runs the entire design flow through the Place & Route process, and then closes the project. A typical script would probably contain more steps, but you can use this example as a general guideline.

```tcl
prj_project open "C:/lscc/Radiant/counter/counter.rdf"
prj_run PAR -impl counter -forceAll
prj_project close
save_script c:/lscc/radiant/examples/counter/myscript2.tcl
```
Running Tcl Scripts

You can run scripts from the Radiant software integrated Tcl Console whether your project is opened or not. You can also run scripts from the external Tcl Console prompt window. The following example procedure uses the integrated Tcl Console and the sample Tcl script from the previous section:

To run a Tcl script that opens a project, runs processes and closes the project:

1. Open the Radiant software but do not open your project. If your project is open, choose File > Close Project.

2. If you are using the Radiant software main window, click the small arrow pane switch in the bottom of the Radiant software main window, and then click on the Tcl Console tab in the Output area at the bottom to open the console.

3. If there are previously issued commands in the console, type reset in the console command line to refresh your session and clear out all previous commands.

```reset```

4. Use the TCL `source` command to load and run your TCL script. Since it’s the only argument, the `source` command requires the filename of the script you want to load and run. Prefix the script file name with any required relative or absolute path information. To run the example script shown in the previous section use the following:

```source C:/lscc/radiant/<version_number>/examples/counter/myscript2.tcl```

5. As long as there are no syntax errors or invalid arguments, the script will open the project, synthesize, map, and place-and-route the design. Once the design finishes it closes the project. If there are errors in the script, you will see the errors in red in the Tcl Console after you attempt to run it. Go back to your script and correct the errors that prevented the script from running.

Project Archiving

A Radiant software project archive is a single compressed file (.zip) of your entire project. The project archive can contain all of the files in your project directory, or it can be limited to source-related files. When you use the File > Archive Project command, the dialog box provides the option to “Archive all files under the Project directory.” When you select this option, the entire project is archived. When you clear this option, only the project’s source-related files, including strategies, are archived. Many of these source-related files must be archived in order to achieve the same bitstream results for a fully implemented design.

Whichever archiving method you select, if your project contains source files stored outside the project folder, the remote files will be compressed under the remote_files subdirectory in the archive; for example:
<project_name>/remote_files/sources
<project_name>/remote_files/strategies

When unarchiving, you must manually move the archived remote files to the original locations or the project will not work.

File Descriptions

This section provides a list of the file types used in the Radiant software, including those generated during design implementation. The Archive column indicates the files that must be archived in order to achieve the same bitstream results.

Table 28: Source Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Definition</th>
<th>Function</th>
<th>Archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>.fdc</td>
<td>FPGA Design Constraint file</td>
<td>Used for specifying design constraints for Synplify-Pro synthesis tool.</td>
<td>✓</td>
</tr>
<tr>
<td>.ipk</td>
<td>Radiant Software IP Package file.</td>
<td>Package file for Radiant software Soft IP.</td>
<td></td>
</tr>
<tr>
<td>.ipx</td>
<td>Manifest file generated by IP Catalog.</td>
<td>Enables changes to be made to a module or IP Catalog.</td>
<td>✓</td>
</tr>
<tr>
<td>.ldc</td>
<td>Lattice Design Constraint file</td>
<td>Used for specifying timing constraints for LSE synthesis flow. The .ldc contents will be combined into design database file .udb.</td>
<td>✓</td>
</tr>
<tr>
<td>.pcf</td>
<td>Power Calculator project file</td>
<td>Stores power analysis results from information extracted from the design project.</td>
<td>✓</td>
</tr>
<tr>
<td>.pdc</td>
<td>Post-Synthesis constraint file.</td>
<td>Used for specifying post-synthesis constraints (timing/physical) for Lattice engines such as MAP and PAR.</td>
<td>✓</td>
</tr>
<tr>
<td>.rdf</td>
<td>Radiant Software Project file</td>
<td>Used for managing and implementing all project files in the Radiant software.</td>
<td>✓</td>
</tr>
<tr>
<td>.rva</td>
<td>Reveal Analyzer file</td>
<td>Defines the Reveal Analyzer project and contains data about the display of signals in Waveform View.</td>
<td>✓</td>
</tr>
<tr>
<td>.rvi</td>
<td>Reveal Inserter debug file</td>
<td>Defines the Reveal project and its modules, identifies the trace and trigger signals, and stores the trace and trigger options.</td>
<td>✓</td>
</tr>
<tr>
<td>.sdc</td>
<td>SDC constraints file</td>
<td>Used for specifying design-specific constraints for Synplify-Pro. SDC is replaced by the FDC format in but is still supported in the Radiant software.</td>
<td>✓</td>
</tr>
<tr>
<td>.spf</td>
<td>Simulation project file, a script file produced by the Simulation Wizard</td>
<td>Used for running the simulator for your project from the Radiant software.</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Table 28: Source Files (Continued)

<table>
<thead>
<tr>
<th>File Type</th>
<th>Definition</th>
<th>Function</th>
<th>Archive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>.sty</td>
<td>Strategy file</td>
<td>Defines the optimization control settings of implementation tools such as logic synthesis, mapping, and place and route.</td>
<td>✓</td>
</tr>
<tr>
<td>.v</td>
<td>Verilog source file</td>
<td>Contains Verilog description of the circuit structure and function.</td>
<td>✓</td>
</tr>
<tr>
<td>.vhd</td>
<td>VHDL source file</td>
<td>Contains VHDL description of the circuit structure and function.</td>
<td>✓</td>
</tr>
<tr>
<td>.xcf</td>
<td>Configuration chain file</td>
<td>Used for programming devices in a JTAG daisy chain.</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Table 29: IP Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Definition</th>
<th>Function</th>
<th>Archive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;instName&gt;_bb.v</td>
<td>Verilog IP black box file</td>
<td>Provides the Verilog synthesis black box for the IP core and defines the port list.</td>
<td>✓</td>
</tr>
<tr>
<td>&lt;instName&gt;.cfg</td>
<td>IP parameter configuration file</td>
<td>Used for re-creating or modifying the core in the IP Catalog tool.</td>
<td>✓</td>
</tr>
<tr>
<td>&lt;instName&gt;.svg</td>
<td>Scalable Vector Graphics file</td>
<td>A graphic file used to display module/IP schematic and ports.</td>
<td>✓</td>
</tr>
<tr>
<td>&lt;instName&gt;_tmpl.v</td>
<td>Verilog template file</td>
<td>A template for instantiating the generated module. This file can be copied into a user Verilog file.</td>
<td>✓</td>
</tr>
<tr>
<td>&lt;instName&gt;_tmpl.vhd</td>
<td>VHDL module template file</td>
<td>A template for instantiating the generated module. This file can be copied into a user VHDL file.</td>
<td>✓</td>
</tr>
<tr>
<td>&lt;instName&gt;.v</td>
<td>Verilog module file</td>
<td>Verilog netlist generated by IP Catalog to match the user configuration of the module.</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Table 30: Implementation Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Definition</th>
<th>Function</th>
<th>Archive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bgn</td>
<td>Bitstream generation report file</td>
<td>Reports results of a bit generation (bitgen) run and displays information on options that were set.</td>
<td></td>
</tr>
<tr>
<td>.bin</td>
<td>Bitstream file</td>
<td>Used for SRAM FPGA programming.</td>
<td></td>
</tr>
<tr>
<td>.ibs</td>
<td>Post-Route I/O buffer information specification file (IBIS)</td>
<td>Used for analyzing signal integrity and electromagnetic compatibility (EMC) on printed circuit boards.</td>
<td></td>
</tr>
<tr>
<td>.mcs</td>
<td>PROM file</td>
<td>Used for SRAM FPGA programming.</td>
<td></td>
</tr>
<tr>
<td>.mrp</td>
<td>Map Report file</td>
<td>Provides statistics about component usage in the mapped design.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 30: Implementation Files (Continued)

<table>
<thead>
<tr>
<th>File Type</th>
<th>Definition</th>
<th>Function</th>
<th>Archive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>.pad</td>
<td>Post-Route PAD report file</td>
<td>Lists all programmable I/O cells used in the design and their associated primary pins.</td>
<td></td>
</tr>
<tr>
<td>.par</td>
<td>Post-Route Place &amp; Route report file</td>
<td>Summarizes information from all iterations and shows the steps taken to achieve a placement and routing solution.</td>
<td></td>
</tr>
<tr>
<td>.sso</td>
<td>Post-PAR SSO analysis file</td>
<td>Reports the noise caused by simultaneously switching outputs.</td>
<td></td>
</tr>
<tr>
<td>.tw1</td>
<td>Post-Map Timing analysis file</td>
<td>Estimates pre-route timing.</td>
<td></td>
</tr>
<tr>
<td>.twr</td>
<td>Post-PAR Timing analysis file</td>
<td>Reports post-route timing.</td>
<td></td>
</tr>
<tr>
<td>.vo</td>
<td>Post-Route Verilog simulation file</td>
<td>Used for post-route simulation.</td>
<td></td>
</tr>
<tr>
<td>&lt;Design name&gt;_vo.sdf</td>
<td>Post-Route SDF simulation file for Verilog</td>
<td>Used for timing simulation.</td>
<td></td>
</tr>
<tr>
<td>.vm</td>
<td>Synthesized netlist file</td>
<td>Netlist file generated by the Radiant software Synthesis tools.</td>
<td></td>
</tr>
<tr>
<td>.udb</td>
<td>Unified Design Database file</td>
<td>Compiled from HDL design source. It may contain both design netlist and constraints.</td>
<td></td>
</tr>
</tbody>
</table>
# Revision History

The following table gives the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/19/2019</td>
<td>2.0</td>
<td>▶ Updated to reflect changes in Radiant 2.0 software.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Removed Appendix A - Reveal User Guide. Starting with Radiant software v2.0, this is now a stand-alone User Guide.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ Removed Appendix B -- Programming Tools User Guide. Starting with Radiant software v2.0, this is now a stand-alone User Guide.</td>
</tr>
<tr>
<td>03/25/2019</td>
<td>1.1</td>
<td>Updated to reflect changes in Radiant 1.1 software.</td>
</tr>
<tr>
<td>02/08/2018</td>
<td>1.0</td>
<td>Initial Release.</td>
</tr>
</tbody>
</table>